СТРУКТУРЫ ДАННЫХ
И АЛГОРИТМЫ

DATA STRUCTURES AMD ALGORITHMS

ALFRED V. AHO
Bell Laboratories
Murray Hill, New Jersey

JOHN E. HOPKROFT
Cornell University
Ithaca, New York

JEFFREY D. ULLMAN
Stanford University
Stanford, California

ADDISON-WESLEY PUBLISHING COMPANY
Reading, Massachusetts ♦ Menlo Park, California
London ♦ Amsterdam ♦ Don Mills, Ontario ♦ Sydney

СТРУКТУРЫ ДАННЫХ И АЛГОРИТМЫ

АЛЬФРЕД В. АХО
Bell Laboratories
Муррей-Хилл, Нью-Джерси

ДЖОН Э. ХОПКРОФТ
Корнеллский университет
Итака, Нью-Йорк

ДЖЕФФРИ Д. УЛЬМАН
Стэнфордский университет
Станфорд, Калифорния



Издательский дом "Вильямс"
Москва • Санкт-Петербург • Киев
2003

ББК 32.973.26-018.2я75
А95
УДК 681.3.07

Издательский дом "Вильямс"
Зав. редакцией С. *Н. Тригуб*

Перевод с английского и редакция *канд. физ.-мат. наук А. А. Минько*

По общим вопросам обращайтесь в Издательский дом "Вильямс"
по адресу: info@williamspublishing.com, http://www.williamspublishing.com

Ахо, Альфред, В., Хопкрофт, Джон, Ульман, Джеффри, Д.
А95 Структуры данных и алгоритмы. : Пер. с англ. : М. : Издательский дом "Вильямс", 2003. — 384 с. : ил. — Парал. тит. англ.
ISBN 5-8459-0122-7 (рус.)

В этой книге подробно рассмотрены структуры данных и алгоритмы, которые являются фундаментом современной методологии разработки программ. Показаны разнообразные реализации абстрактных типов данных, начиная от стандартных списков, стеков, очередей и заканчивая множествами и отображениями, которые используются для неформального описания и реализации алгоритмов. Две главы книги посвящены методам анализа и построения алгоритмов; приведено и исследовано множество различных алгоритмов для работы с графами, внутренней и внешней сортировки, управления памятью.

Книга не требует от читателя специальной подготовки, только предполагает его знакомство с какими-либо языками программирования высокого уровня, такими как Pascal. Вместе с тем она будет полезна специалистам по разработке программ и алгоритмов и может быть использована как учебное пособие для студентов и аспирантов, специализирующихся в области компьютерных наук.

ББК 32.973.26-018.2я75

Издательство выражает признательность Дмитрию Владимировичу Виленскому за информационную поддержку

ISBN 5-8459-0122-7 (рус.)
ISBN 0-201-00023-7 (англ.)

Оглавление

Содержание

Предисловие

В этой книге описаны структуры данных и алгоритмы, которые являются фундаментом современного компьютерного программирования. Основу данной книги составляют первые шесть глав нашей ранее изданной книги *The Design and Analysis of Computer Algorithms*[1]. Мы расширили ее содержание, включив материал по алгоритмам внешнего хранения и управлению памятью. Как и предыдущая, эта книга может составить основу учебного курса по структурам данным и алгоритмам. Мы не требуем от читателя специальной подготовки, только предполагаем его знакомство с какими-либо языками программирования высокого уровня, такими как Pascal.

Мы попытались осветить структуры данных и алгоритмы в более широком контексте решения задач с использованием вычислительной техники, а также использовали абстрактные типы данных для неформального описания и реализации алгоритмов. И хотя сегодня абстрактные типы данных только начинают применять в современных языках программирования, авторы считают, что они являются полезным инструментом при разработке программ независимо от применяемого языка программирования.

Мы также постоянно подчеркиваем и внедряем идею вычисления и оценки времени выполнения алгоритмов (временную сложность алгоритмов) как составную часть процесса компьютерного решения задач. В этом отражается наша надежда на то, что программисты осознают, что при решении задач прогрессирующе больших размеров особое значение имеет временная сложность выбранного алгоритма, а не возможности новых поколений вычислительных средств.

Представление алгоритмов

Мы используем язык Pascal для представления описываемых алгоритмов и структур данных просто потому, что это широко известный язык программирования. В начале книги алгоритмы часто будут представлены как в абстрактной форме, так и на языке Pascal. Это сделано для того, чтобы показать весь спектр проблем при решении практических задач: от проблемы формализации задачи до проблем, возникающих во время выполнения законченной программы. Алгоритмы, которые мы представляем, можно реализовать на любых языках программирования высокого уровня.

Содержание книги

В главе 1 содержатся вводные замечания и обсуждаются реализации процесса *исходная задача — готовая программа* и роль абстрактных типов данных в этом процессе. Здесь также можно познакомиться с математическим аппаратом, необходимым для оценивания времени выполнения алгоритмов.

В главе 2 рассматриваются традиционные структуры списков, стеков и очередей, а также отображения, которые являются абстрактным типом данных, основанным на математическом понятии функции. В главе 3 вводятся деревья и основные структуры данных, которые эффективно поддерживают различные операторы, выполняемые над деревьями.

В главах 4, 5 рассмотрено большое количество абстрактных типов данных, основанных на математической модели множеств. Достаточно глубоко исследованы словари и очереди с приоритетами. Рассмотрены стандартные реализации этих абстрактных типов данных, такие как хеш-таблицы, двоичные деревья по-

---

[1] Существует перевод этой книги на русский язык: *Построение и анализ вычислительных алгоритмов.* — М., "Мир", 1979. — *Прим.ред.*

иска, частично упорядоченные деревья, 2-3 деревья и др. Более сложный материал помещен в главу 5.

Главы 6, 7 содержат материал, относящийся к графам; ориентированные графы рассмотрены в главе 6, а неориентированные — в главе 7. Эти главы начинают раздел книги, который больше посвящен алгоритмам, чем структурам данных, хотя мы продолжаем обсуждать основные структуры данных, подходящие для представления графов. В этих главах представлено большое количество алгоритмов для работы с графами, включая алгоритмы поиска в глубину, нахождения минимального остовного дерева, кратчайших путей и максимальных паросочетаний.

В главе 8 рассмотрены основные алгоритмы внутренней сортировки: быстрая сортировка, пирамидальная сортировка, "карманная" сортировка, а также более простые (и менее эффективные) методы, например метод сортировки вставками. В конце главы описаны алгоритмы с линейным временем выполнения для нахождения медиан и других порядковых статистик.

В главе 9 обсуждаются асимптотические методы анализа рекурсивных программ. Здесь, конечно же, рассмотрены методы решения рекуррентных соотношении.

В главе 10 сравнительно кратко (без глубокого анализа) рассмотрены методы разработки алгоритмов, включая методы декомпозиции, динамическое программирование, алгоритмы локального поиска, и различные формы организации деревьев поиска.

Последние две главы посвящены организации внешнего хранения и управлению памятью. Глава 11 охватывает методы внешней сортировки и организацию внешнего хранения данных, включая В-деревья и индексные структуры.

Материал по управлению памятью, содержащийся в главе 12, условно можно разбить на четыре части, в зависимости от того, являются ли блоки памяти фиксированной или переменной длины, а также от того, явно или неявно осуществляется очистка памяти.

Материал этой книги авторы использовали в программе лекционных курсов по структурам данным и алгоритмам в Колумбийском, Корнеллском и Станфордском университетах как для студентов, так и для аспирантов. Например, предварительную версию этой книги использовали в Станфордском университете как основу 10-недельного курса по структурам данных для студентов первого года обучения. Этот курс включал материал глав 1-4, 9, 10, 12 и частично из глав 5-7.

Упражнения

В конце каждой главы приведено много упражнений различной степени сложности. С помощью многих из них можно просто проверить усвоение изложенного материала. Некоторые упражнения требуют дополнительных усилий и размышлений, они помечены одной звездочкой. Упражнения, помеченные двумя звездочками, рассчитаны на студентов старших курсов и аспирантов. Библиографические примечания в конце каждой главы предлагают ссылки на дополнительную литературу.

Благодарности

Мы хотим выразить благодарность компании Bell Laboratories за предоставление превосходных коммуникационных средств и средств подготовки текстов (основанных на UNIX™), которые позволили легко подготовить рукопись географически разделенным авторам. Многие наши коллеги, прочитав различные части рукописи, сделали ценные замечания. Особенно мы хотим поблагодарить за полезные предложения Эда Бекхама (Ed Beckham), Джона Бентли (Jon Bentley), Кеннета Чу (Kenneth Chu), Джанет Кореи (Janet Coursey), Хенка Кокса (Hank Cox), Нейла Иммермана (Neil Immerman), Брайана Кернигана (Brian Kernighan), Стива Мехени (Steve Mahaney), Крейга Мак-Мюррей (Craig McMurray), Альберто Мендельзона (Alberto Mendelzon), Элистер Моффат (Alistair Moffat), Джеффа Нотона (Jeff Naughton), Керри Нимовичер (Kerry Nemovicher), Пола Ниэмки (Paul Niamkey), Ёшио Оно (Yoshio Ohno), Роба Пайка (Rob Pike), Криса Руэна (Chris Rouen), Мориса Шлумбергера (Maurice Schlumberger), Стенли Селкова (Stanley Selkow), Ченгай Ши (Chengya Shih), Боба Тарьяна (Bob Tarjan), В. Ван Снайдера (W. Van Snyder), Питера Вейнбергера (Peter Weinberger) и Энтони Ерёкериса (Anthony Yeracaris). Наконец, мы хотим принести огромную благодарность г-же Клейр Мецгер (Claire Metzger) за профессиональную помощь при подготовке рукописи к печати.

A.V.A.
J.E.H.
J.D.U.

ГЛАВА 1

Построение и анализ алгоритмов

Процесс создания компьютерной программы для решения какой-либо практической задачи состоит из нескольких этапов: формализация и создание технического задания на исходную задачу; разработка алгоритма решения задачи; написание, тестирование, отладка и документирование программы; получение решения исходной задачи путем выполнения законченной программы, В данной главе мы покажем подходы к реализации этих этапов, а в последующих рассмотрим алгоритмы и структуры данных как строительные блоки создаваемых компьютерных программ.

1.1. От задачи к программе

Половина дела сделана, если знать, что исходная задача имеет решение. В первом приближении большинство задач, встречающихся на практике, не имеют четкого и однозначного описания. Определенные задачи, такие как разработка рецепта вечной молодости или сохранение мира во всем мире, вообще невозможно сформулировать в терминах, допускающих компьютерное решение. Даже если мы предполагаем, что наша задача может быть решена на компьютере, обычно для ее формального описания требуется огромное количество разнообразных параметров. И часто только в ходе дополнительных экспериментов можно найти интервалы изменения этих параметров.

Если определенные аспекты решаемой задачи можно выразить в терминах какой-либо формальной модели, то это, безусловно, необходимо сделать, так как в этом случае в рамках формальной модели мы можем узнать, существуют ли методы и алгоритмы решения нашей задачи. Даже если такие методы и алгоритмы не существуют на сегодняшний день, то привлечение средств и свойств формальной модели поможет в построении "подходящего" решения исходной задачи.

Практически любую область математики или других наук можно привлечь к построению модели определенного круга задач. Для задач, числовых по своей природе, можно построить модели на основе общих математических конструкций, таких как системы линейных уравнений (например, для задач расчета электрических цепей или напряжений в закрепленных балках), дифференциальные уравнения (задачи прогноза роста популяций или расчета скорости протекания химических реакций). Для задач с символьными или текстовыми данными можно применить модели символьных последовательностей или формальных грамматик. Решение таких задач содержит этапы компиляции (преобразование программ, написанных на языке высокого уровня, в программы на машинно-ориентированных языках) и информационного поиска (распознавание определенных слов в списках заголовков каких-либо библиотек и т.п.).

Алгоритмы

Когда построена (подобрана) подходящая модель исходной задачи, то естественно искать решение в терминах этой модели. На этом этапе основная цель заключается в построении решения в форме ***алгоритма***, состоящего из конечной последовательности инструкций, каждая из которых имеет четкий смысл и может быть выполнена с конечными вычислительными затратами за конечное время. Целочисленный оператор присваивания $x := y + z$ — пример инструкции, которая будет выполнена с конечными вычислительными затратами. Инструкции могут выполняться в алгоритме любое число раз, при этом они сами определяют число повторений. Однако мы требуем, чтобы при любых входных данных алгоритм завершился после выполнения конечного числа инструкций. Таким образом, программа, написанная на основе разработанного алгоритма, при любых начальных данных никогда не должна приводить к бесконечным циклическим вычислениям.

Есть еще один аспект определения алгоритмов, о котором необходимо сказать. Выше мы говорили, что алгоритмические инструкции должны иметь "четкий смысл" и выполняться с "конечными вычислительными затратами". Естественно, то, что понятно одному человеку и имеет для него "четкий смысл", может совершенно иначе представляться другому. То же самое можно сказать о понятии "конечных затрат": на практике часто трудно доказать, что при любых исходных данных выполнение последовательности инструкций завершится, даже если мы четко понимаем смысл каждой инструкции. В этой ситуации, учитывая все аргументы за и против, было бы полезно попытаться достигнуть соглашения о "конечных затратах" в отношении последовательности инструкций, составляющих алгоритм. Однако кажущаяся сложность подобных доказательств может быть обманчивой. В разделе 1.5 мы оценим время выполнения основных структур языков программирования, что докажет конечность времени их выполнения и соответственно конечность вычислительных затрат»

Кроме программ на языке Pascal, мы часто будем представлять алгоритмы с помощью *псевдоязыка* программирования, который комбинирует обычные конструкции языков программирования с выражениями на "человеческом" языке. Мы используем Pascal как язык программирования, но практически любой другой язык программирования может быть использован вместо него для представления алгоритмов, рассматриваемых в этой книге.

Следующие примеры иллюстрируют основные этапы создания компьютерной программы.

Пример 1.1. Рассмотрим математическую модель, используемую для управления светофорами на сложном перекрестке дорог. Мы должны создать программу, которая в качестве входных данных использует множество всех допустимых поворотов на перекрестке (продолжение прямой дороги, проходящей через перекресток, также будем считать "поворотом") и разбивает это множество на несколько групп так, чтобы все повороты в группе могли выполняться одновременно, не создавая проблем друг для друга. Затем мы сопоставим с каждой группой поворотов соответствующий режим работы светофоров на перекрестке. Желательно минимизировать число разбиений исходного множества поворотов, поскольку при этом минимизируется количество режимов работы светофоров на перекрестке.

Для примера на рис. 1.1 показан перекресток возле Принстонского университета, известный сложностью его преодоления. Обратите внимание, что дороги С и £ односторонние, остальные — двухсторонние. Всего на этом перекрестке возможно 13 поворотов. Некоторые из этих поворотов, такие как *AB* (поворот с дороги *A* на дорогу *B)* и *EC,* могут выполняться одновременно. Трассы других поворотов, например *AD* и *EB,* пересекаются, поэтому их нельзя выполнять одновременно. Режимы работы светофоров должны учитывать эти обстоятельства и не допускать одновременного выполнения таких поворотов, как *AD* и *EB,* но могут разрешать совместное выполнение поворотов, подобных *AB* и *EC.*

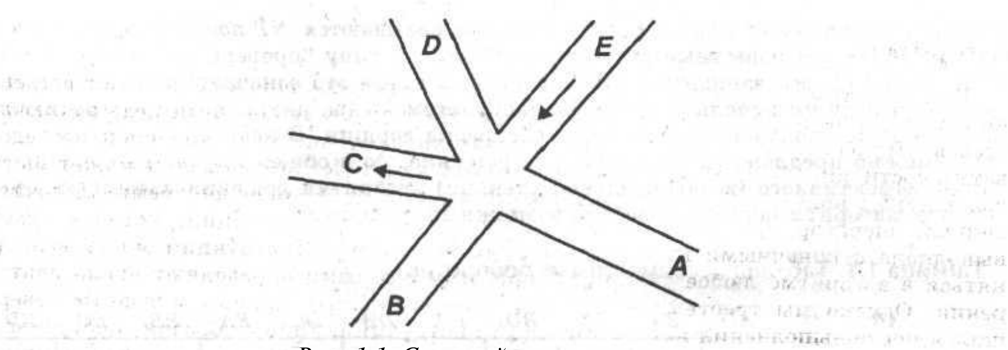


Рис. 1.1. Сложный перекресток

Для построения модели этой задачи можно применить математическую структуру, известную как граф. *Граф* состоит из множества точек, которые называются *вершинами,* и совокупности линий *(ребер),* соединяющих эти точки. Для решения задачи управления движением по перекрестку можно нарисовать граф, где вершины будут представлять повороты, а ребра соединят ту часть вершин-поворотов, которые нельзя выполнить **одновременно.** Для нашего перекрестка (рис. 1.1) соответствующий граф показан на рис. 1.2, а в табл. **1.1** дано другое представление графа — в виде таблицы, где на пересечении строки i и столбца *j* стоит 1 тогда и только тогда, когда существует ребро между вершинами i и *j*.

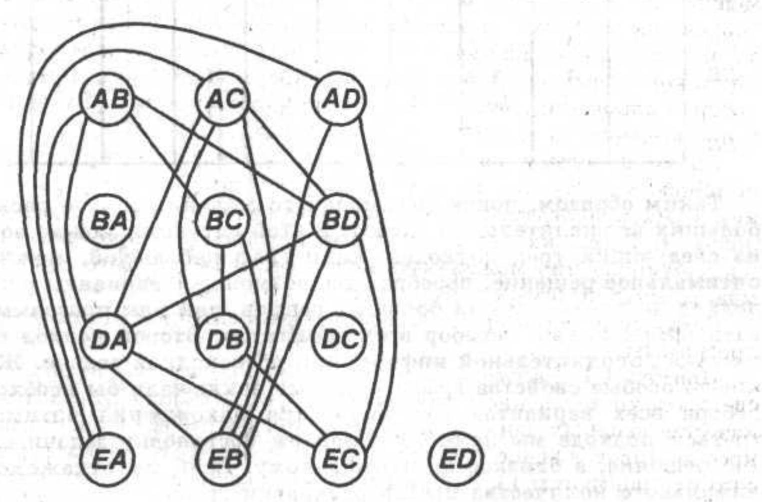


Рис. 1.2. Граф, показывающий несовместимые повороты

Модель в виде графа поможет в решении исходной задачи **управления** светофорами. В рамках этой модели можно использовать решение, которое дает математическая задача *раскраски графа:* каждой вершине графа надо так задать цвет, чтобы никакие две соединенные ребром вершины не имели одинаковый цвет, и при этом по возможности использовать минимальное количество цветов. Нетрудно видеть, что при такой раскраске графа несовместимым поворотам будут соответствовать вершины, окрашенные в разные цвета.

Проблема раскраски графа изучается математиками уже несколько десятилетий, но теория алгоритмов мало что может сказать о решении этой проблемы. К сожалению, задача раскраски произвольного графа минимальным количеством

цветов принадлежит классу задач, которые называются ***NP*-полными** *задачами* и для которых все известные решения относятся к типу "проверь все возможности" или "перебери все варианты". Для раскраски графа это означает, что сначала для закраски вершин используется один цвет, затем — два цвета, потом — три и т.д., пока не будет получена подходящая раскраска вершин. В некоторых частных случаях можно предложить более быстрое решение, йо в общем случае не существует более эффективного (в значительной степени) алгоритма решения задачи раскраски, чем алгоритм полного перебора возможных вариантов.

Таблица 1.1. Таблица несовместимых поворотов

	AB	*AC*	*AD*	*BA*	*BC*	*BD*	*DA*	*DB*	*DC*	*EA*	*EB*	*EC*	*ED*
AB					1	1	1			1			
AC						1	1	1		1	1		
AD										1	1	1	
BA													
BC	1							1			1		
BD	1	1					1				1	1	
DA	1	1				1					1	1	
DB		1			1							1	
DC													
EA	1	1	1										
EB		1	1		1	1	1						
EC			1			1	1	1					
ED													

Таким образом, поиск оптимального решения задачи раскраски графа требует больших вычислительных затрат. В этой ситуации можно воспользоваться одним из следующих трех подходов. Если граф небольшой, можно попытаться найти оптимальное решение, перебрав все возможные варианты раскраски. Однако этот подход не приемлем для больших графов, так как программно трудно организовать эффективный перебор всех вариантов. Второй подход предполагает использование дополнительной информации об исходной задаче. Желательно найти какие-то особые свойства графа, которые исключали бы необходимость полного перебора всех вариантов раскраски для нахождения оптимального решения. В третьем подходе мы немного изменяем постановку задачи и ищем не оптимальное решение, а близкое к оптимальному. Если мы откажемся от требования минимального количества цветов раскраски графа, то можно построить алгоритмы раскраски, которые работают значительно быстрее, чем алгоритмы полного перебора. Алгоритмы, которые быстро находят "подходящее", но не оптимальное решение, называются *эвристическими.*

Примером рационального эвристического алгоритма может служить следующий "жадный" алгоритм раскраски графа. В этом алгоритме сначала мы пытаемся раскрасить как можно больше вершин в один цвет, затем закрашиваем во второй цвет также по возможности максимальное число оставшихся вершин, и т.д. При закраске вершин в новый цвет мы выполняем следующие действия.

1. Выбираем произвольную незакрашенную вершину и назначаем ей новый цвет.
2. Просматриваем список незакрашенных вершин и для каждой из них определяем, соединена ли она ребром с вершиной, уже закрашенной в новый цвет. Если не соединена, то к этой вершине также применяется новый цвет.

Этот алгоритм назван "жадным" из-за того, что каждый цвет применяется к максимально большому числу вершин, без возможности пропуска некоторых из них или перекраски ранее закрашенных. Возможны ситуации, когда, будь алгоритм менее "жадным" и пропустил бы некоторые вершины при закраске новым цветом, мы получили бы раскраску графа меньшим количеством цветов. Например, для раскраски графа на рис. 1.3 можно было бы применить два цвета, закрасив вершину 1 в красный цвет, а затем, пропустив вершину 2, закрасить в красный цвет вершины 3 и 4. Но "жадный" алгоритм, основываясь на порядковой очередности вершин, закрасит в красный цвет вершины 1 и 2, для закраски остальных вершин потребуются еще два цвета.

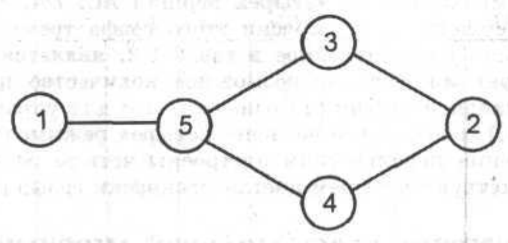


Рис. 1.3. Граф

Применим **описанный** алгоритм для закраски вершин графа нашей задачи **(рис.1.2),** при этом первой закрасим вершину *AB* в синий цвет. Также можно закрасить в синий цвет вершины *AC, AD* и *BA,* поскольку никакие из этих четырех вершин не имеют общих ребер. Вершину *BC* нельзя **закрасить** в этот цвет, так как существует ребро между вершинами *AB* и *BC*. По этой же причине мы не можем закрасить в синий цвет вершины *BD, DA* и *DB* — все они имеют хотя бы по одному ребру, связывающему их с уже закрашенными в синий цвет вершинами. Продолжая перебор вершин, закрашиваем вершины *DC* и *ED* в синий цвет, а вершины *EA, EB* и *EC* оставляем незакрашенными.

Теперь применим второй цвет, например красный. Закраску этим цветом начнем с вершины ВС. Вершину *BD* можно закрасить красным цветом, вершину *DA* — нельзя, так как есть ребро между вершинами *BD* и *DA.* Продолжаем перебор вершин: вершину *DB* нельзя закрасить красным цветом, вершина *DC* уже закрашена синим цветом, а вершину *EA* можно закрасить красным. Остальные незакрашенные вершины имеют общие ребра с вершинами, окрашенными в красный цвет, поэтому к ним нельзя применить этот цвет.

Рассмотрим незакрашенные вершины *DA, DB, EB* и *EC.* Если вершину *DA* мы закрасим в зеленый цвет, то в этот цвет также можно закрасить и вершину *DB,* но вершины *EB* и *EC* в зеленый цвет закрасить нельзя. К последним двум вершинам применим четвертый цвет, скажем, желтый. Назначенные вершинам цвета приведены в табл. 1.2. В этой таблице "дополнительные" повороты совместимы с соответствующими поворотами из столбца "Повороты", хотя алгоритмом они окрашены в разные цвета. При задании режимов работы светофоров, основанных на одном из цветов раскраски, вполне безопасно включить в эти режимы и дополнительные повороты.

Данный алгоритм не всегда использует минимально возможное число цветов. Можно использовать результаты из общей теории графов для оценки качества полученного решения. В теории графов ***k*-кликой** называется множество из k вершин, в котором каждая пара вершин соединена ребром. Очевидно, что для закраски fe-клики необходимо k цветов, поскольку в клике никакие две вершины не могут иметь одинаковый цвет.

Таблица 1.2. Раскраска графа, представленного на рис. 1.2

Цвет	Повороты	Дополнительные повороты
Синий	*AB*, AC, AD, *BA*, *DC*, *ED*	—
Красный	*BC*, *BD*, *EA*	*BA*, *OC*, *ED*
Зеленый	*DA*, *DB*	*AD*, *BA*, *DC*, *ED*
Желтый	*EB*, *EC*	*BA*, *DC*, *EA*, *ED*

В графе рис. 1.2 множество из четырех вершин *AC*, *DA*, *BD* и *EB* является 4-кликой. Поэтому не существует раскраски этого графа тремя и менее цветами, и, следовательно, решение, представленное в табл. 1.2, является оптимальным в том смысле, что использует минимально возможное количество цветов для раскраски графа. В терминах исходной задачи это означает, что для управления перекрестком, показанным на рис. 1.1, необходимо не менее четырех режимов работы светофоров.

Итак, для управления перекрестком построены четыре режима работы светофорами, которые соответствуют четырем цветам раскраски графа (табл. 1.2). □

Псевдоязык и пошаговая „кристаллизация" алгоритмов

Поскольку для решения исходной задачи мы применяем некую математическую модель, то тем самым можно формализовать алгоритм решения в терминах этой модели. В начальных версиях алгоритма часто применяются обобщенные операторы, которые затем переопределяются в виде более мелких, четко определенных инструкций. Например, при описании рассмотренного выше алгоритма раскраски графа мы использовали такие выражения, как "выбрать произвольную незакрашенную вершину". Мы надеемся, что читателю совершенно ясно, как надо понимать такие инструкции. Но для преобразования таких неформальных алгоритмов в компьютерные программы необходимо пройти через несколько этапов формализации (этот процесс можно назвать *пошаговой кристаллизацией),* пока мы не получим программу, полностью состоящую из формальных операторов языка программирования.

Пример 1.2. Рассмотрим процесс преобразования "жадного" алгоритма раскраски графа в программу на языке Pascal. Мы предполагаем, что есть граф *G*, вершины которого необходимо раскрасить. Программа *greedy* (жадный) определяет множество вершин, названное *newclr* (новый цвет), все вершины которого можно окрасить в новый цвет. Эта программа будет вызываться на повторное выполнение столько раз, сколько необходимо для закраски всех вершин исходного графа. В самом первом грубом приближении программа *greedy* на псевдоязыке показана в следующем листинге.

Листинг 1.1. Первое приближение программы *greedy*

```
    procedure greedy ( var G: GRAPH; var newclr: SET );
      { greedy присваивает переменной newclr множество вершин
        графа G, которые можно окрасить в один цвет}
    begin
(1)     newclr:= ∅¹;
(2)     for для каждой незакрашенной вершины v из G do
(3)       if v не соединена с вершинами из newclr then begin
(4)         пометить v цветом;
(5)         добавить v в newclr
          end
    end; { greedy }
```

[1] Символ ∅ — стандартное обозначение пустого множества.

В листинге 1.1 вы легко выделите средства нашего псевдоязыка. Мы применяем полужирное начертание для зарезервированных ключевых слов языка Pascal, которые имеют тот же смысл, что и в стандартном языке Pascal. Написание строчными буквами таких слов, как GRAPH (Граф) и SET[1] (Множество), указывает на имена *абстрактных типов данных*. Их можно определить с помощью объявления типов языка Pascal и операторов, соответствующих абстрактным типам данных, которые задаются посредством процедур языка Pascal (эти процедуры должны входить в окончательный вариант программы). Более детально абстрактные типы данных мы рассмотрим в следующих двух разделах этой главы.

Управляющие конструкции языка Pascal, такие как if, for и while, могут применяться в операторах псевдоязыка, но условные выражения в них (как в строке (3)) могут быть неформальными, в отличие от строгих логических выражений Pascal. Также и в строке (1) в правой части оператора присваивания применяется неформальный символ. Еще отметим, что оператор цикла **for** в строке (2) проводит повторные вычисления по элементам множества.

Чтобы стать исполняемой, программа на псевдоязыке должна быть преобразована в программу на языке Pascal. В данном случае мы не будем приводить все этапы такого преобразования, а покажем только пример преобразования оператора if (строка (3)) в более традиционный код.

Чтобы определить, имеет ли вершина *v* соединение с какой-либо вершиной из *newclr*, мы рассмотрим каждый элемент *w* из *newclr* и по графу *G* проверим, существует ли ребро между вершинами *v* и *w*. Для этого используем новую булеву переменную *found* (поиск), которая будет принимать значение true (истина), если такое ребро существует. Листинг 1.2 показывает частично преобразованную программу листинга 1.1.

Листинг 1.2. Частично преобразованная программа *greedy*

```
        procedure greedy ( var G: GRAPH; var newclr: SET );
            begin
(1)             newclr:= 0;
(2)             for для каждой незакрашенной вершины v из G do begin
(3.1)               found:= false;
(3.2)               for для каждой вершины w из newclr do
(3.3)                   if существует ребро между v и w then
(3.4)                       found:= true;
(3.5)               if found = false then begin
                        { v не соединена ни с одной вершиной из newclr }
(4)                     пометить v цветом;
(5)                     добавить v в newclr
                    end
                end
            end; { greedy }
```

Обратите внимание, что наш алгоритм работает с двумя множествами вершин. Внешний цикл (строки (2) - (5)) выполняется над множеством незакрашенных вершин графа *G*. Внутренний цикл (строки (3.2) - (3.4)) работает с текущим множеством вершин *newclr*. Оператор строки (5) добавляет новые вершины в это множество.

Существуют различные способы представления множеств в языках программирования. В главах 4 и 5 мы изучим несколько таких представлений. В этом примере мы можем представить каждое множество вершин посредством абстрактного типа LIST (Список), который можно выполнить в виде обычного списка целых чисел, ограниченного специальным значением *null* (для обозначения которого мы будем использовать число 0). Эти целые числа могут храниться, например, в массиве, но есть и другие способы представления данных типа LIST, которые мы рассмотрим в главе 2.

[1] Мы отличаем абстрактный тип данных SET от встроенного типа данных set языка Pascal.

Теперь можно записать оператор for в строке (3.2) в виде стандартного цикла по условию, где переменная *w* инициализируется как первый элемент списка *newclr* и затем при каждом выполнении цикла принимает значение следующего элемента из *newclr*. Мы также преобразуем оператор **for** строки (2) листинга 1.1. Измененная процедура *greedy* представлена в листинге 1.3. В этом листинге есть еще операторы, которые необходимо преобразовать в стандартный код языка Pascal, но мы пока ограничимся сделанным. □

Листинг 1.3. Измененная программа *greedy*

```
procedure greedy ( var G: GRAPH; var newclr: LIST );
    { greedy присваивает переменной newclr множество вершин
            графа G, которые можно окрасить в один цвет }
  var
      found: boolean;
      v, w: integer;
  begin
      newclr:= 0;
      v:= первая незакрашенная вершина из G;
      while v <> null do begin
          found:= false;
          w:= первая вершина из newclr
          while w <> null do begin
              if существует ребро между v и w then
                  found:= true;
              w:= следующая вершина из newclr;
          end;
          if found = false then begin
              пометить v цветом;
              добавить v в newclr
          end
          v:= следующая незакрашенная вершина из G;
      end;
  end; { greedy }
```

Резюме

На рис. 1.4 схематически представлен процесс программирования так, как он трактуется в этой книге. На первом этапе создается модель исходной задачи, для чего привлекаются соответствующие подходящие математические модели (такие как теория графов в предыдущем примере). На этом этапе для нахождения решения также строится неформальный алгоритм.

На следующем этапе алгоритм записывается на псевдоязыке — композиции (смеси) конструкций языка Pascal и менее формальных и обобщенных операторов на простом "человеческом" языке. Продолжением этого этапа является замена неформальных операторов последовательностью более детальных и формальных операторов. С этой точки зрения программа на псевдоязыке должна быть достаточно подробной, так как в ней фиксируются (определяются) различные типы данных, над которыми выполняются операторы. Затем создаются абстрактные типы данных для каждого зафиксированного типа данных (за исключением элементарных типов данных, таких как целые и действительные числа или символьные строки) путем задания имен процедурам для каждого оператора, выполняемого над данными абстрактного типа, и замены их (операторов) вызовом соответствующих процедур.

Третий этап процесса программирования обеспечивает реализацию каждого абстрактного типа данных и создание процедур для выполнения различных операторов над данными этих типов. На этом этапе также заменяются все неформальные операторы

псевдоязыка на код языка Pascal. Результатом этого этапа должна быть выполняемая программа. После ее отладки вы получите работающую программу, и мы надеемся, что, используя пошаговый подход при разработке программ, схема которого показана на рис. 1.4, процесс отладки конечной программы будет небольшим и безболезненным.

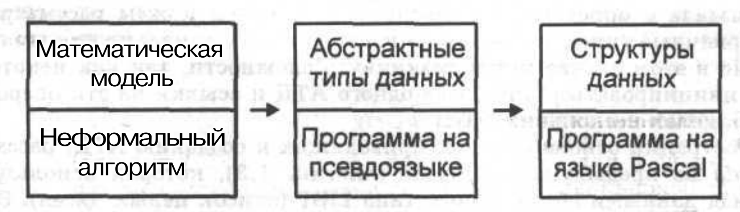


Рис. 1.4. Схема процесса создания программ для решения прикладных задач

1.2. Абстрактные типы данных

Большинство понятий, введенных в предыдущем разделе, обычно излагаются в начальном курсе программирования и должны быть знакомы читателю. Новыми могут быть только абстрактные типы данных, поэтому сначала обсудим их роль в процессе разработки программ. Прежде всего сравним абстрактный тип данных с таким знакомым понятием, как процедура.

Процедуру, неотъемлемый инструмент программирования, можно рассматривать как обобщенное понятие оператора. В отличие от ограниченных по своим возможностям встроенных операторов языка программирования (сложения, умножения и т.п.), с помощью процедур программист может создавать собственные операторы и применять их к операндам различных типов, не только базовым. Примером такой процедуры-оператора может служить стандартная подпрограмма перемножения матриц.

Другим преимуществом процедур (кроме способности создавать новые операторы) является возможность использования их для *инкапсулирования* частей алгоритма путем помещения в отдельный раздел программы всех операторов, отвечающих за определенный аспект функционирования программы. Пример инкапсуляции: использование одной процедуры для чтения входных данных любого типа и проверки их корректности. Преимущество инкапсуляции заключается в том, что мы знаем, какие инкапсулированные операторы необходимо изменить в случае возникновения проблем в функционировании программы. Например, если необходимо организовать проверку входных данных на положительность значений, следует изменить только несколько строк кода, и мы точно знаем, где эти строки находятся.

Определение абстрактного типа данных

Мы определяем *абстрактный тип данных* (АТД) как математическую модель с совокупностью операторов, определенных в рамках этой модели. Простым примером АТД могут служить множества целых чисел с операторами объединения, пересечения и разности множеств. В модели АТД операторы могут иметь операндами не только данные, определенные АТД, но и данные других типов: стандартных типов языка программирования или определенных в других АТД. Результат действия оператора также может иметь тип, отличный от определенных в данной модели АТД. Но мы предполагаем, что по крайней мене один операнд или результат любого оператора имеет тип данных, определенный в рассматриваемой модели АТД.

Две характерные особенности процедур — обобщение и инкапсуляция, — о которых говорилось выше, отлично характеризуют абстрактные типы данных. АТД можно рассматривать как обобщение простых типов данных (целых и действительных чисел и т.д.), точно так же, как процедура является обобщением простых операторов (+, - и т.д.). АТД инкапсулирует типы данных в том смысле, что определение типа и

все операторы, выполняемые над данными этого типа, помещаются в один раздел программы. Если необходимо изменить реализацию АТД, мы знаем, где найти и что изменить в одном небольшом разделе программы, и можем быть уверенными, что это не приведет к ошибкам где-либо в программе при работе с этим типом данных. Более того, вне раздела с определением операторов АТД мы можем рассматривать типы АТД как первичные типы, так как объявление типов формально не связано с их реализацией. Но в этом случае могут возникнуть сложности, так как некоторые операторы могут инициироваться для более одного АТД и ссылки на эти операторы должны быть в разделах нескольких АТД.

Для иллюстрации основных идей, приводящих к созданию АТД, рассмотрим процедуру *greedy* из предыдущего раздела (листинг 1.3), которая использует простые операторы над данными абстрактного типа LIST (список целых чисел). Эти операторы должны выполнить над переменной ***newclr*** типа LIST следующие действия.

1. Сделать список пустым.
2. Выбрать первый элемент списка и, если список пустой, возвратить значение ***null***.
3. Выбрать следующий элемент списка и возвратить значение *null*, если следующего элемента нет.
4. Вставить целое число в список.

Возможно применение различных структур данных, с помощью которых можно эффективно выполнить описанные действия. (Подробно структуры данных будут рассмотрены в главе 2.) Если в листинге 1.3 заменить соответствующие операторы выражениями

MAKENULL(*newclr*);
w := FIRST(*newclr*);
w := NEXT(*newclr*);
INSERT(*v*, *newclr*);

то будет понятен один из основных аспектов (и преимуществ) абстрактных типов данных. Можно реализовать тип данных любым способом, а программы, использующие объекты этого типа, не зависят от способа реализации типа — за это отвечают процедуры, реализующие операторы для этого типа данных.

Вернемся к абстрактному типу данных GRAPH (Граф). Для объектов этого типа необходимы операторы, которые выполняют следующие действия.

1. Выбирают первую незакрашенную вершину.
2. Проверяют, существует ли ребро между двумя вершинами.
3. Помечают вершину цветом.
4. Выбирают следующую незакрашенную вершину.

Очевидно, что вне поля зрения процедуры *greedy* остаются и другие операторы, такие как вставка вершин и ребер в граф или помечающие все вершины графа как незакрашенные. Различные структуры данных, поддерживающие этот тип данных, будут рассмотрены в главах 6 и 7.

Необходимо особо подчеркнуть, что количество операторов, применяемых к объектам данной математической модели, не ограничено. Каждый набор операторов определяет отдельный АТД. Вот примеры операторов, которые можно определить для абстрактного типа данных SET (Множество).

1. MAKENULL(*A*). Эта процедура делает множество *A* пустым множеством.
2. UNION(*A*, *B*, *C*). Эта процедура имеет два "входных" аргумента, множества *A* и *B*, и присваивает объединение этих множеств "выходному" аргументу — множеству *C*.
3. SIZE(*A*). Эта функция имеет аргумент-множество *A* и возвращает объект целого типа, равный количеству элементов множества *A*.

Термин *реализация* АТД подразумевает следующее: перевод в операторы языка программирования объявлений, определяющие переменные этого абстрактного типа данных, плюс процедуры для каждого оператора, выполняемого над объектами АТД. Реализация зависит от *структуры данных,* представляющих АТД. Каждая структура данных строится на основе базовых типов данных применяемого языка программирования, используя доступные в этом языке средства структурирования данных. Структуры массивов и записей — два важных средства структурирования данных, возможных в языке Pascal. Например, одной из возможных реализаций переменной S типа SET может служить массив, содержащий элементы множества S.

Одной из основных причин определения двух различных АТД в рамках одной модели является то, что над объектами этих АТД необходимо выполнять различные действия, т.е. определять операторы разных типов. В этой книге рассматривается только несколько основных математических моделей, таких как теория множеств и теория графов, но при различных реализациях на основе этих моделей определенных АТД будут строиться различные наборы операторов.

В идеале желательно писать программы на языке, базовых типов данных и операторов которого достаточно для реализации АТД. С этой точки зрения язык Pascal не очень подходящий язык для реализации различных АТД, но, с другой стороны, трудно найти иной язык программирования, в котором можно было бы так непосредственно декларировать АТД. Дополнительную информацию о таких языках программирования см. в библиографических примечаниях в конце главы.

1.3. Типы данных, структуры данных и абстрактные типы данных

Хотя термины *тип данных* (или просто *тип), структура данных* и *абстрактный тип данных* звучат похоже, но имеют они различный смысл. В языках программирования *тип данных* переменной обозначает множество значений, которые может принимать эта переменная. Например, переменная булевого (логического) типа может принимать только два значения: значение true (истина) и значение false (ложь) и никакие другие. Набор базовых типов данных отличается в различных языках: в языке Pascal это типы целых (integer) и действительных (real) чисел, булев (boolean) тип и символьный (char) тип. Правила конструирования составных типов данных (на основе базовых типов) также различаются в разных языках программирования: как мы уже упоминали, Pascal легко и быстро строит такие типы.

Абстрактный тип данных — это математическая модель плюс различные операторы, определенные в рамках этой модели. Как уже указывалось, мы можем разрабатывать алгоритм в терминах АТД, но для реализации алгоритма в конкретном языке программирования необходимо найти способ представления АТД в терминах типов данных и операторов, поддерживаемых данным языком программирования. Для представления АТД используются *структуры данных,* которые представляют собой набор переменных, возможно, различных типов данных, объединенных определенным образом.

Базовым строительным блоком структуры данных является *ячейка,* которая предназначена для хранения значения определенного базового или составного типа данных. Структуры данных создаются путем задания имен совокупностям (агрегатам) ячеек и (необязательно) интерпретации значения некоторых ячеек как представителей (т.е. указателей) других ячеек.

В качестве простейшего механизма агрегирования ячеек в Pascal и большинстве других языков программирования можно применять (одномерный) массив, т.е. последовательность ячеек определенного типа. Массив также можно рассматривать как отображение множества индексов (таких как целые числа 1, 2, ..., га) в множество ячеек. Ссылка на ячейку обычно состоит из имени массива и значения из множества индексов данного массива. В Pascal множество индексов может быть нечислового ти-

па, например (север, восток, юг, запад), или интервального типа (как **1..10).** Значения всех ячеек массива должны иметь одинаковый тип данных. Объявление

```
имя: array[ТипИндекса] of ТипЯчеек;
```

задает *имя* для последовательности ячеек, тип для элементов множества индексов и тип содержимого ячеек.

Кстати, Pascal необычайно богат на типы индексов. Многие языки программирования позволяют использовать в качестве индексов только множества последовательных целых чисел. Например, чтобы в языке Fortran в качестве индексов массива можно было использовать буквы, надо все равно использовать целые индексы, заменяя "А" на 1, "В" на 2, и т.д.

Другим общим механизмом агрегирования ячеек в языках программирования является *структура записи.* ***Запись*** (record) можно рассматривать как ячейку, состоящую из нескольких других ячеек (называемых *полями),* значения в которых могут быть разных типов. Записи часто группируются в массивы; тип данных определяется совокупностью типов полей записи. Например, в Pascal объявление

```
var
    reclist: array[1..4] of record
        data: real;
        next: integer
    end
```

задает имя *reclist* (список записей) 4-элементного массива, значениями которого являются записи с двумя полями: *data* (данные) и *next* (следующий).

Третий метод агрегирования ячеек, который можно найти в Pascal и некоторых других языках программирования, — это *файл.* Файл, как и одномерный массив, является последовательностью значений определенного типа. Однако файл не имеет индексов: его элементы доступны только в том порядке, в каком они были записаны в файл. В отличие от файла, массивы и записи являются структурами с "произвольным доступом", подразумевая под этим, что время доступа к компонентам массива или записи не зависит от значения индекса массива или указателя поля записи. Достоинство агрегирования с помощью файла (частично компенсирующее описанный недостаток) заключается в том, что файл не имеет ограничения на количество составляющих его элементов и это количество может изменяться во время выполнения программы.

Указатели и курсоры

В дополнение к средствам агрегирования ячеек в языках программирования можно использовать указатели и курсоры. *Указатель* (pointer) — это ячейка, чье значение указывает на другую ячейку. При графическом представлении структуры данных в виде схемы тот факт, что ячейка *A* является указателем на ячейку *B,* показывается с помощью стрелки от ячейки *A* к ячейке *B.*

В языке Pascal с помощью следующего объявления можно создать переменную-указатель ***prt***, указывающую на ячейку определенного типа, например ТипЯчейки:

```
var
    prt: ↑ ТипЯчейки
```

Постфикс в виде стрелки, направленной вверх, в Pascal используется как оператор разыменования, т.е. выражение ***prt***↑ обозначает значение (типа ТипЯчейки) в ячейке, указанной *prt.*

Курсор (cursor) — это ячейка с целочисленным значением, используемая для указания на массив. В качестве способа указания курсор работает так же, как и указатель, но курсор можно Использовать и в языках (подобных Fortran), которые не имеют явного типа указателя. Интерпретировав целочисленную ячейку как индекс-

ное значение для **массива,** можно эффективно реализовать указания на ячейки массива. К сожалению, этот прием подходит только для ячеек массива и не позволяет организовать указание на ячейки, не являющиеся частью массива.

В схемах структур данных мы будем рисовать стрелку из ячейки курсора к ячейке, на которую указывает курсор. Иногда мы также будем показывать целое число в ячейке курсора, напоминая тем самым, что это не настоящий указатель. Читатель может заметить, что механизм указателя Pascal разрешает ячейкам массива только "быть указанными" с помощью курсора, но не быть истинным указателем. Другие языки программирования, подобные **PL/1** или С, позволяют компонентам массивов быть истинными указателями и, конечно, "быть указанным" с помощью курсора. В отличие от этих языков, в языках Fortran и Algol, где нет типа указателя, можно использовать только курсоры.

Пример 1.3. На рис. 1.5 показана структура данных, состоящая из двух частей. Она имеет цепочку ячеек, содержащих курсоры для массива ***reclist*** (список записей), определенного выше. Назначение поля *next* (следующий) заключается в указании на следующую запись в массиве *reclist*. Например, ***reclist*[4].*next*** равно 1, поэтому запись 4 предшествует записи 1. Полагая первой запись 4, в соответствии со значениями поля *next* получим следующий порядок записей: 4, 1, 3, 2. Отметим, что значение поля *next* в записи 2, равное 0, указывает на то, что нет следующей записи. Целесообразно принять соглашение, что число 0 будет обозначать ***нуль-указатель*** при использовании курсоров и указателей. Но, чтобы не возникали проблемы при реализации этого соглашения, необходимо также **условиться,** что массивы, на которые указывают курсоры, индексируются начиная с 1, а не с 0.

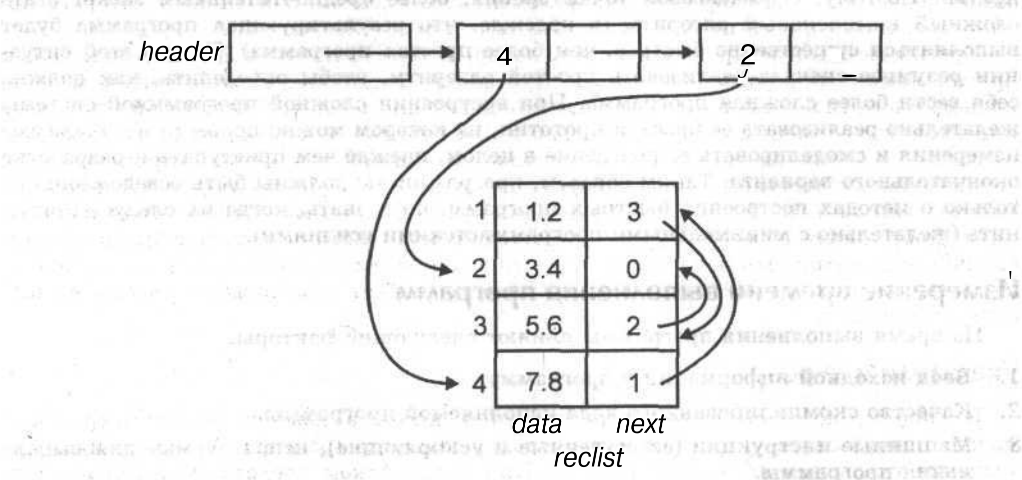


Рис. 1.5. Пример структуры данных

Ячейки в цепочке на рис. 1.5 имеют тип

```
type
    recordtype = record
        cursor: integer;
        prt: t recordtype
    end
```

На цепочку можно **ссылаться** с помощью переменной *header* (заголовок), имеющей тип T recordtype, *header* указывает на анонимную запись типа **recordtype**[1].

---

[1] Запись **анонимна** (не имеет имени), так как создается с помощью вызова ***new*(*header*)**, где *header* указывает на вновь создаваемую запись. В памяти компьютера, конечно, есть адрес, по которому можно локализовать ячейку с записью.

Эта запись имеет значение 4 в поле *cursor*. Мы рассматриваем 4 как индекс массива **reclist**. Эта запись имеет истинный указатель в поле *ptr* на другую анонимную запись, которая, в свою очередь, указывает на индекс 4 массива *reclist* и имеет нуль-указатель в поле *prt*. □

1.4. Время выполнения программ

В процессе решения прикладных задач выбор подходящего алгоритма вызывает определенные трудности. В самом деле, на чем основывать свой выбор, если алгоритм должен удовлетворять следующим противоречащим друг другу требованиям.

1. Быть простым для понимания, перевода в программный код и отладки.
2. Эффективно использовать компьютерные ресурсы и выполняться по возможности быстро.

Если написанная программа должна выполняться только несколько раз, то первое требование наиболее важно. Стоимость рабочего времени программиста обычно значительно превышает стоимость машинного времени выполнения программы, поэтому стоимость программы оптимизируется по стоимости написания (а не выполнения) программы. Если мы имеем дело с задачей, решение которой требует значительных вычислительных затрат, то стоимость выполнения программы может превысить стоимость написания программы, особенно если программа должна выполняться многократно. Поэтому, с финансовой точки зрения, более предпочтительным может стать сложный комплексный алгоритм (в надежде, что результирующая программа будет выполняться существенно быстрее, чем более простая программа). Но и в этой ситуации разумнее сначала реализовать простой алгоритм, чтобы определить, как должна себя вести более сложная программа. При построении сложной программной системы желательно реализовать ее простой прототип, на котором можно провести необходимые измерения и смоделировать ее поведение в целом, прежде чем приступать к разработке окончательного варианта. Таким образом, программисты должны быть осведомлены не только о методах построения быстрых программ, но и знать, когда их следует применить (желательно с минимальными программистскими усилиями).

Измерение времени выполнения программ

На время выполнения программы влияют следующие факторы.

1. Ввод исходной информации в программу.
2. Качество скомпилированного кода исполняемой программы.
3. Машинные инструкции (естественные и ускоряющие), используемые для выполнения программы.
4. Временная сложность алгоритма соответствующей программы.[1]

Поскольку время выполнения программы зависит от ввода исходных данных, его можно определить как функцию от исходных данных. Но зачастую время выполнения программы зависит не от самих исходных данных, а от их "размера". В этом отношении хорошим примером являются задачи *сортировки*, которые мы подробно рассмотрим в главе 8. В задачах сортировки в качестве входных данных выступает

---

[1] Здесь авторы, не акцентируя на этом внимания и не определяя его особо, вводят термин "временная сложность алгоритма". Под *временной сложностью алгоритма* понимается "время" выполнения алгоритма, измеряемое в "шагах" (инструкциях алгоритма), которые необходимо выполнить алгоритму для достижения запланированного результата. В качестве синонима для этого термина авторы часто используют выражение "время выполнения алгоритма". Отметим, что в первой части книги (до главы 6) авторами чаще используется последнее выражение, а во второй — чаще термин "временная сложность". В этой связи заметим, что необходимо различать "время выполнения алгоритма" и "время выполнения программы". — *Прим. ред.*

список элементов, подлежащих сортировке, а в качестве выходного результата — те же самые элементы, отсортированные в порядке возрастания или убывания. Например, входной список 2, 1, 3, 1, 5, 8 будет преобразован в выходной список 1, 1, 2, 3, 5, 8 (в данном случае список *отсортирован в порядке возрастания)*. Естественной мерой объема входной информации для программы сортировки будет число элементов, подлежащих сортировке, или, другими словами, длина входного списка, В общем случае длина входных данных — Подходящая мера объема входной информации, и если не будет оговорено иное, то в качестве меры объема входной информации мы далее будем понимать именно длину входных данных.

Обычно говорят, что время выполнения программы имеет порядок $T(n)$ от входных данных размера n. Например, некая программа имеет время выполнения $T(n) = cn^2$, где c — константа. Единица измерения $T(n)$точно не определена, но мы будем понимать $T(n)$как количество инструкций, выполняемых на идеализированном компьютере.

Для многих программ время выполнения действительно является функцией входных данных, а не их размера. В этой ситуации мы определяем $T(n)$ как время выполнения *в наихудшем случае,* т.е. *как* максимум времени выполнения по всем входным данным размера n. Мы также будем рассматривать $T_{ср}(n)$ как среднее (в статистическом смысле) время выполнения по всем входным данным размера n. Хотя $T_{ср}(n)$является достаточно объективной мерой времени выполнения, но часто, нельзя предполагать (или обосновать) равнозначность всех входных данных. На практике среднее время выполнения найти сложнее, чем наихудшее время выполнения, так как математически это трудноразрешимая задача и, кроме того, зачастую не имеет простого определения понятие "средних" входных данных. Поэтому в основном мы будем использовать наихудшее время выполнения как меру временной сложности алгоритмов, но не будем забывать и о среднем времени выполнения там, где это возможно,

Теперь сделаем замечание о втором и третьем факторах, влияющих на время выполнения программ: о компиляторе, используемом для компиляции программы, и машине, на которой выполняется программа. Эти факторы влияют на то, что для измерения времени выполнения *T(n)* мы не можем применить стандартные единицы измерения, такие как секунды или миллисекунды. Поэтому мы можем только делать заключения, подобные "время выполнения такого-то алгоритма пропорционально n^2". Константы пропорциональности также нельзя точно определить, поскольку они зависят от компилятора, компьютера и других факторов.

Асимптотические соотношения

Для описания скорости роста функций используется O-символика. Например, когда мы говорим, что время выполнения $T(n)$ некоторой программы имеет порядок $O(n^2)$ (читается "о-большое от га в квадрате" или просто "о от *n* в квадрате"), то подразумевается, что существуют Положительные константы c и n_0 такие, что для всех га, больших или равных га$_0$, выполняется неравенство $T(n) < cn^2$.

Пример 1.4. Предположим, что T(0) = 1, $T(1) = 4$ и в общем случае $T(n) =$ (га + 1)^2. Тогда $T(n)$ имеет порядок $O(n^2)$: если положить га$_0$ = 1 и $c = 4$, то легко показать, что для га ≥ 1 будет выполняться неравенство $(n + 1)^2 \leq 4n^2$. Отметим, что нельзя положить га$_0$ = 0, так как $T(0) = 1$ и, следовательно, это значение при любой константе с больше cO^2 = 0. П

Подчеркнем: мы предполагаем, что все функции времени выполнения определены на множестве неотрицательных целых чисел и их значения также неотрицательны, но необязательно целые. Будем говорить, что $T(n)$ имеет порядок $O(f(n))$,если существуют константы с и n_0 такие, что для всех л > га$_0$ выполняется неравенство $T(n) < cf(n)$. Для программ, у которых время выполнения имеет порядок $O(f(n))$, говорят, что они имеют *порядок* (или *степень) роста* $f(n)$.

Пример 1.5. Функция *T(n)* = 3л^3 + $2n^2$ имеет степень роста $O(n^3)$. Чтобы это показать, надо положить га$_0$ = 0 и $c = 5$, так как легко видеть, что для всех целых я ≥ 0 выполняется неравенство $3n^3 + 2n^2 < 5n^3$. Можно, конечно, сказать, что $T(n)$ имеет Порядок $O(n^4)$, но это более слабое утверждение, чем то, что $T(n)$ имеет порядок роста $O(n\ )$.

В качестве следующего примера докажем, что функция 3^n не может иметь порядок $O(2^n)$. **Предположим,** что существуют константы с и n_0 такие, что для всех $n > n_0$ выполняется неравенство 3" < с2". Тогда $c \ge (3/2)^n$ для всех л > n_0. Но (3/2)" принимает любое, как угодно большое, значение при достаточно большом *n*, поэтому не существует такой константы с, которая могла бы мажорировать (3/2)" для всех *n*. П

Когда мы говорим, что $T(n)$ имеет степень роста $O(f(n))$, то подразумевается, что $f(n)$ является верхней границей скорости роста $T(n)$. Чтобы указать нижнюю границу скорости роста $T(n)$, используется обозначение: $T(n)$ есть $\Omega(g(n))$ (читается "омега-большое от $g(n)$" или просто "омега от $g(n)$"), это подразумевает существование такой константы *с*, что бесконечно часто (для бесконечного числа значений *n*) выполняется неравенство $T(n) > cg(n)$.[1]

Пример 1.6. Для проверки того, что $T(n) = n^3 + 2n^2$ есть $\Omega(n^3)$, достаточно положить $c = 1$. Тогда $T(n) > cn^3$ для га = О, 1,... .

Для другого примера положим, что $T(n) = n$ для нечетных га > 1 и $T(n) — n^2/100$ — для четных $n > 0$. Для доказательства того, что $T(n)$есть $\Omega(n^2)$, достаточно положить $c = 1/100$ и рассмотреть множество четных чисел га = 0, 2, 4, 6,... . D

Ограниченность показателя степени роста

Итак, мы предполагаем, что программы можно оценить с помощью функций времени выполнения, пренебрегая при этом константами пропорциональности. С этой точки зрения программа с временем выполнения $O(n^2)$, например, лучше программы с временем выполнения $O(n^3)$. Константы пропорциональности зависят не только от используемых компилятора и компьютера, но и от свойств самой программы. Пусть при определенной комбинации компилятор-компьютер одна программа выполняется за $100n^2$ миллисекунд, а вторая — за $5n^3$ миллисекунд. Может ли вторая программа быть предпочтительнее, чем первая?

Ответ на этот вопрос зависит от размера входных данных программ. При размере входных данных га < 20 программа с временем выполнения $5n^3$ завершится быстрее, чем программа с временем выполнения $100n^2$. Поэтому, если программы в основном выполняются с входными данными небольшого размера, предпочтение необходимо отдать программе с временем выполнения $O(n^3)$. Однако при возрастании га отношение времени выполнения $5n^3/100n^2$ = л/20 также растет. Поэтому при больших л программа с временем выполнения $O(n^2)$ становится предпочтительнее программы с временем выполнения $O(n^3)$. Если даже при сравнительно небольших л, когда время выполнения обеих программ примерно одинаково, выбор лучшей программы представляет определенные затруднения, то естественно для большей надежности сделать выбор в пользу программы с меньшей степенью роста.

Другая причина, заставляющая отдавать предпочтение программам с наименьшей степенью роста времени выполнения, заключается в том, что чем меньше степень роста, тем больше размер задачи, которую можно решить на компьютере. Другими словами, если увеличивается скорость вычислений компьютера, то растет также и размер задач, решаемых на компьютере. Однако незначительное увеличение скорости вычислений компьютера приводит только к небольшому увеличению размера задач, решаемых в течение фиксированного промежутка времени, исключением из этого правила являются программы с низкой степенью роста, как $O(n)$ и $O(n \log n)$.

Пример 1.7. На рис. 1.6 показаны функции времени выполнения (измеренные в секундах) для четырех программ с различной временной сложностью для одного и

---

[1] Отметим асимметрию в определениях *О*- и Ω-символики. Такая асимметрия бывает полезной, когда алгоритм работает быстро на достаточно большом подмножестве, но не на всем множестве входных данных. Например, есть алгоритмы, которые работают значительно быстрее, если длина входных данных является простым числом, а не (к примеру) четным числом. В этом случае невозможно получить хорошую нижнюю границу времени выполнения, справедливую для всех л > *по.*

того же сочетания компилятор-компьютер. Предположим, что можно использовать 1 000 секунд (примерно 17 минут) машинного времени для решения задачи. Какой максимальный размер задачи, решаемой за это время? За 10^3 секунд каждый из четырех алгоритмов может решить задачи примерно одинакового размера, как показано во втором столбце табл. 1.3.

Предположим, что получен новый компьютер (без дополнительных финансовых затрат), работающий в десять раз быстрее. Теперь за ту же цену можно использовать 10^4 секунд машинного времени — ранее 10^3 секунд. Максимальный размер задачи, которую может решить за это время каждая из четырех программ, показан в третьем столбце табл. 1.3. Отношения значений третьего и второго столбцов приведены в четвертом столбце этой таблицы. Здесь мы видим, что увеличение скорости компьютера на 1 000% приводит к увеличению только на 30% размера задачи, решаемой с помощью программы с временем выполнения $O(2^n)$. Таким образом, 10-кратное увеличение производительности компьютера дает в процентном отношении значительно меньший эффект увеличения размера решаемой задачи. В действительности, независимо от быстродействия компьютера, программа с временем выполнения $O(2^n)$ может решать только очень небольшие задачи.

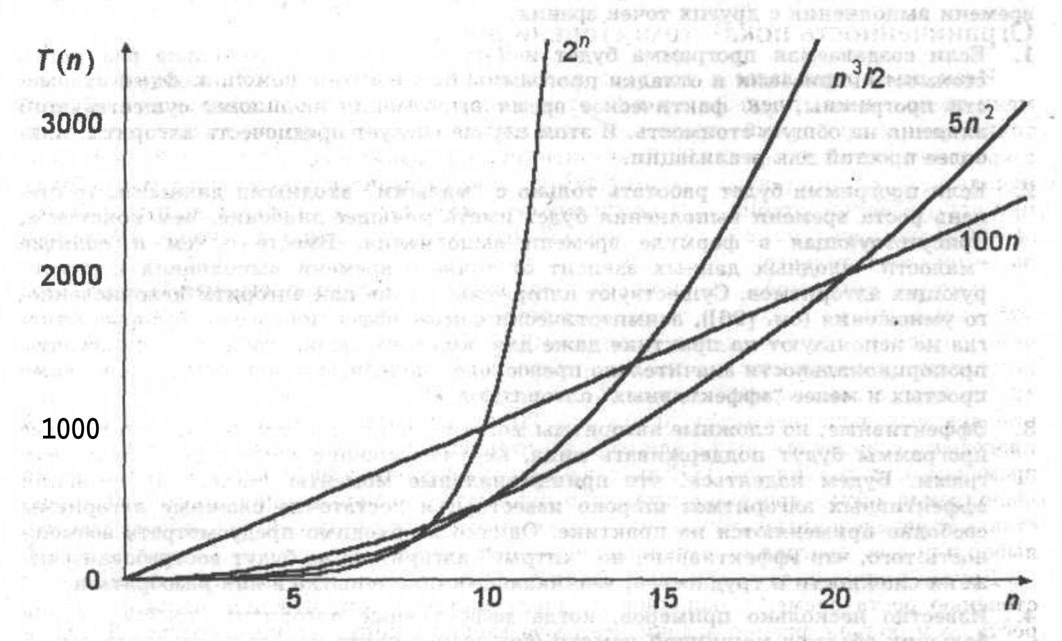


Рис. 1.6. Функции времени выполнения четырех программ

Таблица 1.3. Эффект от 10-кратного увеличения быстродействия компьютера

Время выполнения T(n)	Максимальный размер задачи для 10^3 секунд	Максимальный размер задачи для 10^4 секунд	Увеличение максимального размера задачи
$100n$	10	100	10.0
$5n^2$	14	45	3.2
$n^3/2$	12	27	2.3
2^n	10	13	1.3

Из третьего столбца табл. 1.3 ясно видно преимущество программ с временем выполнения $O(n)$: 10-кратное увеличение размера решаемой задачи при 10-кратном увеличении производительности компьютера. Программы с временем выполнения $O(n^3)$ и $O(n^2)$ при увеличении быстродействия компьютера на 1 000% дают увеличение размера задачи соответственно на 230% и 320%. Эти соотношения сохранятся и при дальнейшем увеличении производительности компьютера. D

Поскольку существует необходимость решения задач все более увеличивающегося размера, мы приходим к почти парадоксальному выводу. Так как машинное время все время дешевеет, а компьютеры становятся более быстродействующими, мы надеемся, что сможем решать все большие по размеру и более сложные задачи. Но вместе с тем возрастает значимость разработки и использования эффективных алгоритмов именно с низкой степенью роста функции времени выполнения.

Немного соли

Мы хотим еще раз подчеркнуть, что степень роста наихудшего времени выполнения — не единственный или самый важный критерий оценки алгоритмов и программ. Приведем несколько соображений, позволяющих посмотреть на критерий времени выполнения с других точек зрения.

1. Если создаваемая программа будет использована только несколько раз, тогда стоимость написания и отладки программы будет доминировать в общей стоимости программы, т.е. фактическое время выполнения не окажет существенного влияния на общую стоимость. В этом случае следует предпочесть алгоритм, наиболее простой для реализации.
2. Если программа будет работать только с "малыми" входными данными, то степень роста времени выполнения будет иметь меньшее значение, чем константа, присутствующая в формуле времени выполнения. Вместе с тем и понятие "малости" входных данных зависит от точного времени выполнения конкурирующих алгоритмов. Существуют алгоритмы, такие как алгоритм целочисленного умножения (см. [96]), асимптотически самые эффективные, но которые никогда не используют на практике даже для больших задач, так как их константы пропорциональности значительно превосходят подобные константы других, более простых и менее "эффективных" алгоритмов.
3. Эффективные, но сложные алгоритмы могут быть нежелательными, если готовые программы будут поддерживать лица, не участвующие в написании этих программ. Будем надеяться, что принципиальные моменты технологии создания эффективных алгоритмов широко известны, и достаточно сложные алгоритмы свободно применяются на практике. Однако необходимо предусмотреть возможность того, что эффективные, но "хитрые" алгоритмы не будут востребованы из-за их сложности и трудностей, возникающих при попытке в них разобраться.
4. Известно несколько примеров, когда эффективные алгоритмы требуют таких больших объемов машинной памяти (без возможности использования более медленных внешних средств хранения), что этот фактор сводит на нет преимущество "эффективности" алгоритма.
5. В численных алгоритмах точность и устойчивость алгоритмов не менее важны, чем их временная эффективность.

1.5. Вычисление времени выполнения программ

Теоретическое нахождение времени выполнения программ (даже без определения констант пропорциональности) — сложная математическая задача. Однако на практике определение времени выполнения (также без нахождения значения констант) является вполне разрешимой задачей — для этого нужно знать только несколько базовых принципов. Но прежде чем представить эти принципы, рассмотрим, как выполняются операции сложения и умножения с использованием O-символики.

Пусть $T_1(n)$ и $T_2(n)$ — время выполнения двух программных фрагментов P_1 и P_2, $T_1(n)$ имеет степень роста $O(f(n))$, а $T_2(n)$ — $O(g(n))$. Тогда $T_1(n) + T_2(n)$, т.е. время последовательного выполнения фрагментов P_1 и P_2, имеет степень роста $O(\max(f(n), g(n)))$. Для доказательства этого вспомним, что существуют константы c_1, c_2, n_1 и n_2 такие, что при $n > n_1$ выполняется неравенство $T_1(n) < c_1 f(n)$, *и*, аналогично, $T_2(n) < c_2 g(n)$, если $n > n_2$. Пусть $n_0 = \max(n_1, n_2)$. Если $n > n_0$, то, очевидно, что $T_1(n) + T_2(n) < c_1 f(n) + c_2 g(n)$. Отсюда вытекает, что при $n > n_0$ справедливо неравенство $T_1(n) + T_2(n) < (c_1 + c_2)\max(f(n), g(n))$. Последнее неравенство и означает, что $T_1(n) + T_2(n)$ имеет порядок роста $O(\max(f(n), g(n)))$.

Пример 1.8. *Правило сумм,* данное выше, используется для вычисления времени последовательного выполнения программных фрагментов с циклами и ветвлениями. Пусть есть три фрагмента с временами выполнения соответственно $O(n^2)$, $O(n^3)$ и $O(n \log n)$. Тогда время последовательного выполнения первых двух фрагментов имеет порядок $O(\max(n^2, n^3))$, т.е. $O(n^3)$. Время выполнения всех трех фрагментов имеет порядок $O(\max(n^3, n \log n))$, это то же самое, что $O(n^3)$. П

В общем случае время выполнения конечной последовательности программных фрагментов, без учета констант, имеет порядок фрагмента с наибольшим временем выполнения. Иногда возможна ситуация, когда порядки роста времен нескольких фрагментов *несоизмеримы* (ни один из них не больше, чем другой, но они и не равны). Для примера рассмотрим два фрагмента с временем выполнения $O(f(n))$ и $O(g(n))$, где

$$f(n) = \begin{cases} n^4, & \text{если л четное;} \\ n^2, & \text{если л нечетное,} \end{cases} \qquad g(n) = \begin{cases} n^2, & \text{если л четное;} \\ n^3, & \text{если л нечетное.} \end{cases}$$

В данном случае правило сумм можно применить непосредственно и получить время выполнения $O(\max(f(n), g(n))$, *т.е.* n^4 при n четном и n^3, если n нечетно.

Из правила сумм также следует, что если $g(n) < f(n)$ для всех л, превышающих n_0, то выражение $O(f(n) + g(n))$ эквивалентно $O(f(n))$. Например, $O(n^2 + n)$ то же самое, что $O(n^2)$.

Правило произведений заключается в следующем. Если $T_1(n)$ и $T_2(n)$ имеют степени роста $O(f(n))$ и $O(g(n))$ соответственно, то произведение $T_1(n)T_2(n)$ имеет степень роста $O(f(n)g(n))$. Читатель может самостоятельно доказать это утверждение, используя тот же подход, который применялся при доказательстве правила сумм. Из правила произведений следует, что $O(cf(n))$ эквивалентно $O(f(n))$, если c — положительная константа. Например, $O(n^2/2)$ эквивалентно $O(n^2)$.

Прежде чем переходить к общим правилам анализа времени выполнения программ, рассмотрим простой пример, иллюстрирующий процесс определения времени выполнения.

Пример 1.9. Рассмотрим программу сортировки *bubble* (пузырек), которая упорядочивает массив целых чисел в возрастающем порядке методом "пузырька" (листинг 1.4). За каждый проход внутреннего цикла (операторы (3)–(6)) "пузырек" с наименьшим элементом "всплывает" в начало массива.

Листинг 1.4. Сортировка методом „пузырька“

```
    procedure bubble ( var A: array [1..n] of integer );
        { Процедура упорядочивает массив А в возрастающем порядке }
    var
        i, j, temp: integer;
      begin
(1)       for i:= 1 to n - 1 do
(2)           for j:= n downto i + 1 do
(3)               if A[j - 1] > A[j] then begin
                      { перестановка местами A[j-1] и A[j] }
(4)                   temp:= A[j - 1];
```

```
(5)                      A[j - 1]:= A[j];
(6)                      A[j]:= temp;
                     end
      end; { bubble }
```

Число элементов n, подлежащих сортировке, может служить мерой объема входных данных. Сначала отметим, что все операторы присваивания имеют некоторое постоянное время выполнения, независящее от размера входных данных. Таким образом, операторы (4) - (6) имеют время выполнения порядка $O(1)$. Запись $O(1)$ означает "равнозначно некой константе". В соответствии с правилом сумм время выполнения этой группы операторов равно $O(\max(1, 1, 1)) = O(1)$.

Теперь мы должны подсчитать время выполнения условных и циклических операторов. Операторы if и for вложены друг в друга, поэтому мы пойдем от внутренних операторов к внешним, последовательно определяя время выполнения условного оператора и каждой итерации цикла. Для оператора if проверка логического выражения занимает время порядка O(1). Мы не знаем, будут ли выполняться операторы в теле условного оператора (строки (4) - (6)), но поскольку мы ищем наихудшее время выполнения, то, естественно, предполагаем, что они выполняются. Таким образом, получаем, что время выполнения группы операторов (3) - (6) имеет порядок $O(1)$.

Далее рассмотрим группу (2) - (6) операторов внутреннего цикла. Общее правило вычисления времени выполнения цикла заключается в суммировании времени выполнения каждой итерации цикла. Для операторов (2) - (6) время выполнения на каждой итерации имеет порядок $O(1)$. Цикл выполняется $n - i$ раз, поэтому по правилу произведений общее время выполнения цикла имеет порядок $O((n - i) \times 1)$, что равно $O(n - i)$.

Теперь перейдем к внешнему циклу, который содержит все исполняемые операторы программы. Оператор (1) выполняется $n - 1$ раз, поэтому суммарное время выполнения программы ограничено сверху выражением

$$\sum_{i=1}^{n-1}(n-i) = n(n-1)/2 = n^2/2 - n/2,$$

которое имеет порядок $O(n^2)$. Таким образом, программа "пузырька" выполняется за время, пропорциональное квадрату числа элементов, подлежащих упорядочиванию. В главе 8 мы рассмотрим программы с временем выполнения порядка $O(n \log n)$, которое существенно меньше $O(n^2)$, поскольку при больших n $\log n$[1] значительно меньше n. П

Перед формулировкой общих правил анализа программ позвольте напомнить, что нахождение точной верхней границы времени выполнения программ только в редких случаях так же просто, как в приведенном выше примере, в общем случае эта задача является интеллектуальным вызовом исследователю. Поэтому не существует исчерпывающего множества правил анализа программ. Мы можем дать только некоторые советы и проиллюстрировать их с разных точек зрения примерами, приведенными в этой книге.

Теперь дадим несколько правил анализа программ. В общем случае время выполнения оператора или группы операторов можно параметризовать с помощью размера входных данных и/или одной или нескольких переменных. Но для времени выполнения программы в целом допустимым параметром может быть только n, размер входных данных.

1. Время выполнения операторов присваивания, чтения и записи обычно имеет порядок $O(1)$. Есть несколько исключений из этого правила, например в языке PL/1, где можно присваивать большие массивы, или в любых других языках, допускающих вызовы функций в операторах присваивания.
2. Время выполнения последовательности операторов определяется с помощью правила сумм. Поэтому степень роста времени выполнения последовательности опе-

[1] Если не указано другое, то будем считать, что все логарифмы определены по основанию 2. Однако отметим, что выражение $O(\log n)$ не зависит от основания логарифма, поскольку $\log_a n = c \log_b n$, где $c = \log_a b$.

раторов без определения констант пропорциональности совпадает с наибольшим временем выполнения оператора в данной последовательности.

3. Время выполнения условных операторов состоит из времени выполнения условно исполняемых операторов и времени вычисления самого логического выражения. Время вычисления логического выражения обычно имеет порядок $O(1)$. Время для всей конструкции **if-then-else** состоит из времени вычисления логического выражения и наибольшего из времени, необходимого для выполнения операторов, исполняемых при значении логического выражения true (истина) и при значении false (ложь).
4. Время выполнения цикла является суммой времени всех исполняемых итераций цикла, в свою очередь состоящих из времени выполнения операторов тела цикла и времени вычисления условия прекращения цикла (обычно последнее имеет порядок $O(1)$). Часто время выполнения цикла вычисляется, пренебрегая определением констант пропорциональности, как произведение количества выполненных итераций цикла на наибольшее возможное время выполнения операторов тела цикла. Время выполнения каждого цикла, если в программе их несколько, должно определяться отдельно.

Вызовы процедур

Для программ, содержащих несколько процедур (среди которых нет рекурсивных), можно подсчитать общее время выполнения программы путем последовательного нахождения времени выполнения процедур, начиная с той, которая не имеет вызовов других процедур. (Вызов процедур мы определяем по наличию оператора call.) Так как мы предположили, что все процедуры нерекурсивные, то должна существовать хотя бы одна процедура, не имеющая вызовов других процедур. Затем можно определить время выполнения процедур, вызывающих эту процедуру, используя уже вычисленное время выполнения вызываемой процедуры. Продолжая этот процесс, найдем время выполнения всех процедур и, наконец, время выполнения всей программы.

Если есть рекурсивные процедуры, то нельзя упорядочить все процедуры таким образом, чтобы каждая процедура вызывала только процедуры, время выполнения которых подсчитано на предыдущем шаге. В этом случае мы должны с каждой рекурсивной процедурой связать временную функцию $T(n)$, где n определяет объем аргументов процедуры. Затем мы должны получить *рекуррентное соотношение* для $T(n)$, т.е. уравнение (или неравенство) для $T(n)$, где участвуют значения $T(k)$ для различных значений k.

Техника решения рекуррентных соотношений зависит от вида этих соотношений, некоторые приемы их решения будут показаны в главе 9. Сейчас же мы проанализируем простую рекурсивную программу.

Пример 1.10. В листинге 1.5 представлена программа вычисления факториала $n!$, т.е. вычисления произведения целых чисел от 1 до n включительно.

Листинг 1.5. Рекурсивная программа вычисления **факториала**

```
        function fact ( n: integer ): integer;
            { fact(n) вычисляет n! }
            begin
(1)             if n <= 1 then
(2)                 fact:= 1
                else
(3)                 fact:= n * fact(n - 1)
            end; { fact }
```

Естественной мерой объема входных данных для функции *fact* является значение n. Обозначим через $T(n)$ время выполнения программы. Время выполнения для строк

(1) и (2) имеет порядок $O(1)$, а для строки (3) — $O(1) + T(n - 1)$. Таким образом, для некоторых констант c и d имеем

$$T(n) = \begin{cases} c + T(n-1), & \text{если } n > 1, \\ d, & \text{если } n < 1. \end{cases} \tag{1.1}$$

Полагая, что га > 2, и раскрывая в соответствии с соотношением (1.1) выражение $T(n - 1)$ (т.е. подставляя в (1.1) $n - 1$ вместо n), получим $T(n) = 2c + T(n - 2)$. Аналогично, если $n > 3$, раскрывая $T(n - 2)$, получим $T(n) = 3c + T(n - 3)$. Продолжая этот процесс, в общем случае для некоторого i, $n > i$, имеем $T(n) = ic + T(n - i)$. Положив в последнем выражении $i = n - 1$, окончательно получаем

$$T(n) = c(n-1) + T(1) = c(n-1) + d. \tag{1.2}$$

Из (1.2) следует, что $T(n)$ имеет порядок $O(n)$. Отметим, что в этом примере анализа программы мы предполагали, что операция перемножения двух целых чисел имеет порядок $O(1)$. На практике, однако, программу, представленную в листинге 1.5, нельзя использовать для вычисления факториала при больших значений n, так как размер получаемых в процессе вычисления целых чисел может превышать длину машинного слова. П

Общий метод решения рекуррентных соотношений, подобных соотношению из примера 1.10, состоит в последовательном раскрытии выражений $T(k)$ в правой части уравнения (путем подстановки в исходное соотношение k вместо n) до тех пор, пока не получится формула, у которой в правой части отсутствует T (как в формуле (1.2)). При этом часто приходится находить суммы различных последовательностей; если значения таких сумм нельзя вычислить точно, то для сумм находятся верхние границы, что позволяет, в свою очередь, получить верхние границы для $T(n)$.

Программы с операторами безусловного перехода

При анализе времени выполнения программ мы неявно предполагали, что все ветвления в ходе выполнения процедур осуществлялись с помощью условных операторов и операторов циклов. Мы останавливаемся на этом факте, так как определяли время выполнения больших групп операторов путем применения правила сумм к этом группам. Однако операторы безусловного перехода (такие как goto) могут порождать более сложную логическую групповую структуру. В принципе, операторы безусловного перехода можно исключить из программы. Но, к сожалению, язык Pascal не имеет операторов досрочного прекращения циклов, поэтому операторы перехода по необходимости часто используются в подобных ситуациях.

Мы предлагаем следующий подход для работы с операторами безусловного перехода, выполняющих выход из циклов, который гарантирует отслеживание всего цикла. Поскольку досрочный выход из цикла скорее всего осуществляется после выполнения какого-нибудь логического условия, то мы можем предположить, что это условие никогда не выполняется. Но этот консервативный подход никогда не позволит уменьшить время выполнения программы, так как мы предполагаем полное выполнение цикла. Для программ, где досрочный выход из цикла не исключение, а правило, консервативный подход значительно переоценивает степень роста времени выполнения в наихудшем случае. Если же переход осуществляется к ранее выполненным операторам, то в этом случае вообще нельзя игнорировать оператор безусловного перехода, поскольку такой оператор создает петлю в выполнении программы, что приводит в нарастанию времени выполнения всей программы.

Нам не хотелось бы, чтобы у читателя сложилось мнение о невозможности анализа времени выполнения программ, использующих операторы безусловного перехода, создающих петли. Если программные петли имеют простую структуру, т.е. о любой паре петель можно сказать, что они или не имеют общих элементов, или одна петля вложена в другую, то в этом случае можно применить методы анализа времени выполнения, описанные в данном разделе. (Конечно, бремя ответственности за пра-

вильное определение структуры петли ложится на того, кто проводит анализ.) Мы без колебаний рекомендуем применять описанные методы для анализа программ, написанных на таких языках, как Fortran, где использование операторов безусловного перехода является естественным делом, но для анализируемых программ допустимы петли только простой структуры.

Анализ программ на псевдоязыке

ЕСЛИ мы знаем степень роста времени выполнения операторов, записанных с помощью неформального "человеческого" языка, то, следовательно, сможем проанализировать программу на псевдоязыке (такие программы будем называть псевдопрограммами). Однако часто мы не можем в принципе знать время выполнения той части псевдопрограммы, которая еще не полностью реализована в формальных операторах языка программирования. Например, если в псевдопрограмме имеется неформальная часть, оперирующая абстрактными типами данных, то общее время выполнение программы в значительной степени зависит от способа реализации АТД. Это не является недостатком псевдопрограмм, так как одной из причин написания программ в терминах АТД является возможность выбора реализации АТД, в том числе по критерию времени выполнения.

При анализе псевдопрограмм, содержащих операторы языка программирования и вызовы процедур, имеющих неформализованные фрагменты (такие как операторы для работы с АТД), можно рассматривать общее время выполнения программы как функцию пока не определенного времени выполнения таких процедур. Время выполнения этих процедур можно параметризовать с помощью "размера" их аргументов. Так же, как и "размер входных данных", "размер" аргументов — это предмет для обсуждения и дискуссий. Часто выбранная математическая модель АТД подсказывает логически обоснованный размер аргументов. Например, если АТД строится на основе множеств, то часто количество элементов множества можно принять за параметр функции времени выполнения. В последующих главах вы встретите много примеров анализа времени выполнения псевдопрограмм.

1.6. Практика программирования

Приведем несколько рекомендаций и соображений, которые вытекают из нашего практического опыта построения алгоритмов и реализации их в виде программ. Некоторые из этих рекомендаций могут показаться очевидными или даже банальными, но их полезность можно оценить только при решении реальных задач, а не исходя из теоретических предпосылок. Читатель может проследить применение приведенных рекомендаций при разработке программ в этой книге, а также может проверить их действенность в собственной практике программирования.

1. *Планируйте этапы разработки программы.* Мы описывали в разделе 1.1 этапы разработки программы: сначала черновой набросок алгоритма в неформальном стиле, затем псевдопрограмма, далее — последовательная формализация псевдопрограммы, т.е. переход к уровню исполняемого кода. Эта стратегия организует и дисциплинирует процесс создания конечной программы, которая будет простой в отладке и в дальнейшей поддержке и сопровождении.
2. *Применяйте инкапсуляцию.* Все процедуры, реализующие АТД, поместите в одно место программного листинга. В дальнейшем, если возникнет необходимость изменить реализацию АТД, можно будет корректно и без особых затрат внести какие-либо изменения, так как все необходимые процедуры локализованы в одном месте программы.
3. *Используйте и модифицируйте уже существующие программы.* Один из неэффективных подходов к процессу программирования заключается в том, что каждый новый проект рассматривается "с нуля", без учета уже существующих про-

грамм. Обычно среди программ, реализованных на момент начала проекта, можно найти такие, которые если решают не всю исходную задачу, то хотя бы ее часть. После создания законченной программы полезно оглянуться вокруг и посмотреть, где еще ее можно применить (возможно, вариант ее применения окажется совсем непредвиденным).

4. *Станьте "кузнецом" инструментов.* На языке программистов *инструмент* (tool) — это программа с широким спектром применения. При создании программы подумайте, нельзя ли ее каким-либо образом обобщить, т.е. сделать более универсальной (конечно, с минимальными программистскими усилиями). Например, предположим, что вам необходимо написать программу, составляющую расписание экзаменов. Вместо заказанной программы можно написать программу-инструмент, раскрашивающий вершины обычного графа (по возможности минимальным количеством цветов) таким образом, чтобы любые две вершины, соединенные ребром, были закрашены в разные цвета. В контексте расписания экзаменов вершины графа — это классы, цвета — время проведения экзаменов, а ребра, соединяющие две вершины-класса, обозначают, что в этих классах экзамены принимает одна и та же экзаменационная комиссия. Такая программа раскраски графа вместе с подпрограммой перевода списка классов в множество вершин графа и цветов в заданные временные интервалы проведения экзаменов составит расписание экзаменов. Программу раскраски можно использовать для решения задач, совсем не связанных с составлением расписаний, например для задания режимов работы светофоров на сложном перекрестке, как было показано в разделе 1.1.

5. *Программируйте на командном уровне.* Часто бывает, что в библиотеке программ не удается найти программу, необходимую для выполнения именно нашей задачи, но мы можем адаптировать для этих целей ту или иную программу-инструмент. Развитые операционные системы предоставляют программам, разработанным для различных платформ, возможность совместной работы в сети вовсе без модификации их кода, за исключением списка команд операционной системы. Чтобы сделать команды компонуемыми, как правило, необходимо, чтобы каждая из них вела себя как *фильтр,* т.е. как программа с одним входным и одним выходным файлом. Отметим, что можно компоновать любое количество фильтров, и, если командный язык операционной системы достаточно интеллектуален, достаточно просто составить список команд в том порядке, в каком они будут востребованы программой.

Пример 1.11. В качестве примера рассмотрим программу *spell,* написанную Джонсоном (S.C. Johnson) с использованием команд UNIX[1]. На вход этой программы поступает файл /1, состоящий из текста на английском языке, на выходе получаем все слова из f_1, не совпадающие со словами из небольшого словаря[2]. Эта программа воспринимает имена и правильно написанные слова, которых нет в словаре, как орфографические ошибки. Но обычно выходной список ошибок достаточно короткий, поэтому его можно быстро пробежать глазами и определить, какие слова в нем не являются ошибками.

Первый фильтр, используемый программой *spell,* — это команда *translate,* которая имеет соответствующие параметры и заменяет прописные буквы на строчные, пробелы — на начало новых строк, а остальные символы оставляет без изменений. На выходе этой команды мы получаем файл f_2, состоящий из тех же слов, что и файл /1, но каждое слово расположено в отдельной строке. Далее выполняется команда sort, упорядочивающая строки входного файла в алфавитном порядке. Результатом выполнения этой команды является файл f_3, содержащий отсортированный список (возможно, с повторениями) слов из файла f_2. Затем команда *unique* удаляет

[1] UNIX — торговая марка Bell Laboratories.

[2] Мы могли бы использовать полный словарь, но многие орфографические ошибки совпадают со словами, которые почти никогда не применяются.

повторяющиеся строки в своем входном файле f_3, создавая выходной файл f_4, содержащий слова из исходного файла (без прописных букв и **повторений**), упорядоченные в алфавитном порядке. Наконец, к файлу f_4 применяется команда *diff*, имеющая параметр, который указывает на файл f_5, содержащий в алфавитном порядке слова из **словаря**, расположенные по одному в строке. Результатом этой команды будет список слов из файла f_4, которые не совпадают со словами из файла f_5, т.е. те слова из исходного списка, которых нет в словаре. Программа *spell* состоит из следующей последовательности команд:

```
spell:  translate [A-Z] → [a-z], пробел → новая строка
        sort
        unique
        diff  словарь
```

Командный уровень программирования требует дисциплины от команды программистов. Они должны писать программы как фильтры везде, где это возможно, и создавать программы-инструменты вместо узкоспециализированных программ всегда, когда для этого есть условия. Существенным вознаграждением за это будет значительное повышение вашего коэффициента отношения результата к затраченным усилиям. □

1.7. Расширение языка Pascal

Большинство программ в этой книге написаны на языке Pascal. Чтобы сделать листинги программ более читабельными, мы иногда будем применять три конструкции, которые вы не найдете в стандартном Pascal, но которые легко трансформируются в операторы языка Pascal. Одна из таких конструкций — нечисловые метки. Если в программе необходимо использовать несколько меток, то нечисловые метки значительно облегчат понимание программы. Например, оператор goto *выход*, очевидно, более понятен, чем goto 561. Для преобразования программы с нечисловыми метками в "чистую" программу Pascal, надо заменить все нечисловые метки различными числами и затем объявить их метками (label) в начале **программы**. Этот процесс можно полностью автоматизировать.

Другой нестандартной конструкцией является оператор возврата return, который позволяет писать более понятные программы без использования операторов goto. Оператор возврата будем использовать в следующей форме: **return(*выражение*)**, где скобка *(выражение)* необязательна. Преобразовать процедуру с операторами возврата в стандартную программу Pascal очень просто. Сначала надо объявить новую метку, скажем 999, и затем поместить ее на последний оператор end процедуры. Например, оператор **return(*x*)** в некой функции *zap* заменяется следующим блоком:

```
begin
    zap:= x;
    goto 999
end
```

Оператор return, не имеющий аргументов, просто заменяется на оператор goto 999.

Пример 1.12. В листинге 1.6 представлена программа, вычисляющая факториал, в которой использованы операторы возврата. В листинге 1.7 показана та же программа после замены операторов return стандартными операторами Pascal. □

Листинг 1.6. Программа вычисления факториала с операторами возврата

```
function fact ( n: integer ): integer;
    begin
        if n <= 1 then
```

```
        return(1)
    else
        return(n * fact(n - 1))
end; { fact }
```

Листинг 1.7. Программа вычисления факториала на языке Pascal

```
function fact ( n: integer ): integer;
    label
        999;
    begin
        if n <= 1 then
            begin
                fact:= 1;
                goto 999
            end
        else
            begin
                fact:= n * fact(n - 1);
                goto 999
            end
    999;
    end; { fact }
```

Третье расширение языка Pascal, которое мы будем применять, заключается в постоянном использовании выражений, описывающих типы, в качестве имен типов. Например, выражения, подобные ↑ celltype, мы считаем допустимыми в качестве имен типов, но недопустимыми для типов параметров процедур или значений, возвращаемых функциями. Технически Pascal требует, чтобы мы использовали имена для типов данных, например prtocell (указатель на ячейку). В этой книге мы будем допускать такие описывающие выражения в качестве имен типов, надеясь, что читатель сможет придумать имя для типа данных и механически заменить описывающее тип выражение именем типа.[1] Другими словами, мы будем использовать операторы, подобные

```
function zap( A: array[1..10] of integer ): ↑ celltype
```

подразумевая под этим следующий оператор (здесь arrayofinteger обозначает "массив целых чисел"):

```
function zap( A: arrayofinteger ): prtocell ,
```

Наконец, замечание о применяемых нами соглашениях в отношении листингов программ. Зарезервированные слова Pascal выделяются полужирным начертанием, типы данных пишутся обычным начертанием, а имена процедур, функций и переменных — курсивом. Мы различаем строчные и прописные буквы.

Упражнения

1.1. В состязаниях футбольной лиги участвуют шесть команд: Соколы, Львы, Орлы, Бобры, Тигры и Скунсы. Соколы уже сыграли с Львами и Орлами. Львы также сыграли с Бобрами и Скунсами. Тигры сыграли с Орлами и Скунсами. Каждая команда играет одну игру в неделю. Найдите расписание матчей, чтобы все команды сыграли по одному разу друг с другом в течение минимального количества недель. *Совет.* Создайте граф, где вершины будут соответство-

[1] Чтобы по возможности не плодить в листингах смесь "английского с нижегородским", мы будем применять русский язык только в псевдопрограммах, оставляя названия типов данных на английском языке и давая их перевод или в тексте, или иногда непосредственно в листингах. — *Прим. ред.*

вать парам команд, которые еще не играли между собой. Что должны обозначать ребра в таком графе, если при правильной раскраске графа каждый цвет соответствует матчам, сыгранным в течение определенной недели?

*1.2. Рассмотрим руку робота с закрепленным одним концом. Рука состоит из двух "колен", каждое из которых может поворачивать руку на 90 градусов вверх и вниз в вертикальной плоскости. Какую математическую модель следует применить для описания перемещения оконечности руки? Разработайте алгоритм для перемещения оконечности руки из одного возможного положения в другое.

*1.3. Предположим, что необходимо перемножить четыре матрицы действительных чисел M_1 x M_2 x M_3 x M_4, где M_1 имеет размер 10 x 20, размер M_2 составляет 20 x 50, M_3 — 50 x 1 и M_4 — 1 × 100. Предположим, что для перемножение двух матриц размером p x q и q x r требуется pqr скалярных операций, это условие выполняется в обычных алгоритмах перемножения матриц. Найдите оптимальный порядок перемножения матриц, который минимизирует общее число скалярных операций. Как найти этот оптимальный порядок в случае произвольного числа матриц?

**1.4. Предположим, что мы хотим разделить множество значений квадратных корней целых чисел от 1 до 100 на два подмножества так, чтобы суммы чисел обоих подмножества были по возможности максимально близки. Если мы имеем всего две минуты машинного времени для решения этой задачи, то какие вычисления необходимо выполнить?

1.5. Опишите "жадный" алгоритм для игры в шахматы. Каковы, по вашему мнению, его достоинства и недостатки?

1.6. В разделе 1.2 мы рассмотрели абстрактный тип данных SET (Множество) с операторами MAKENULL, UNION и SIZE. Для определенности положим, что все множества являются подмножествами множества {0, 1, ..., 31}, а АТД SET интерпретируется как тип данных языка Pascal set of 0..31. Напишите процедуры на языке Pascal для выполнения этих операторов, используя данную реализацию АТД SET.

1.7. *Наибольшим общим делителем* двух целых чисел p и q называется наибольшее целое число d, которое делит p и q нацело. Мы хотим создать программу для вычисления наибольшего общего делителя двух целых чисел p и q, используя следующий алгоритм. Обозначим через r остаток от деления p на q. Если r равно 0, то q является наибольшим общим делителем. В противном случае положим p равным q, затем — q равным r и повторим процесс нахождения остатка от деления p на q.

а) покажите, что этот алгоритм правильно находит наибольший общий делитель;

б) запишите алгоритм в виде программы на псевдоязыке;

в) преобразуйте программу на псевдоязыке в программу на языке Pascal.

1.8. Мы хотим создать программу форматирования текста, которая выравнивала бы текст по ширине строк. Программа должна иметь буфер слов и буфер строки. Первоначально оба буфера пусты. Очередное слово считывается в буфер слов. Если в буфере строки достаточно места, то слово переносится в буфер строки. В противном случае добавляются пробелы между словами до полного заполнения буфера строки, затем после печати строки этот буфер освобождается.

а) запишите алгоритм в виде программы на псевдоязыке;

б) преобразуйте программу на псевдоязыке в программу на языке Pascal.

1.9. Рассмотрим множество из n городов и таблицу расстояний между ними. Напишите программу на псевдоязыке для нахождения кратчайшего пути, который предусматривает посещение каждого города только один раз и возвращается в тот город, откуда начался путь. На сегодняшний день единственным методом точного решения этой задачи является только метод полного перебора

всех **возможных** вариантов. Попытайтесь построить эффективный алгоритм для решения этой задачи, используя эвристический подход.

1.10. Рассмотрим следующие функции от л:

$$/,<я) = n^2;$$

$$f_2(n) = n^2 + 1000л;$$

$$f_3(n) = \begin{cases} n, & \text{если л нечетно,} \\ n^3, & \text{если л четно;} \end{cases}$$

$$f_3(n) = \begin{cases} n, & \text{если } n \le 100, \\ n^3, & \text{если л}>100. \end{cases}$$

Укажите для каждой пары функций, когда $f_i(n)$ имеет порядок $O(f_i(n))$ и когда /,(л) есть $\Omega(f_j(n))$.

1.11. Рассмотрим следующие функции от п:

$$g_1(n) = \begin{cases} n^2 & \text{для четных л > O,} \\ n^3 & \text{для нечетных л} \ge 1; \end{cases}$$

$$g_2(n) = \begin{cases} \text{л} & \text{для } 0 < \text{л} < 100, \\ n^3 & \text{для л} > 100; \end{cases}$$

$$g_3(n) = n^{2.5}.$$

Укажите для каждой пары функций, когда $g_i(n)$ имеет порядок $O(g_j(n))$ и когда $g_i(n)$ есть $\Omega(g_j(n))$.

1.12. **Найдите,** используя **O-символику,** время выполнения в наихудшем случае следующих процедур как функции от л:

а)
```
procegure matmpy ( n: integer);
var
    i, j,  k: integer;
begin
    for i:= 1 to п do
        for j := 1 to л do begin
            C[i,j]:= 0;
            for k:= 1 to л do
                C[i,j]:= C[i,j] + A[i,k] * B[k,j]
        end
end
```

б)
```
procegure mystery ( л: integer ) ;
var
    i, j, k: integer;
begin
    for i : = 1 to n - i do
    for j:= i + 1 to л do
        for k:= 1 to j do
            {группа операторов с временем выполнения O(1) }
end
```

в)
```
procegure veryodd ( л: integer ) ;
var
    i/ j/ x, y: integer;
begin
```

```
    for i:= 1 to л do
        if нечетное(i) then begin
            for j:= 1 to л do
                x:= x + 1;
            for j:= 1 to i do
                y:= y + 1
        end
    end
*г) procegure recursive ( n: integer ): integer;
    begin
        if n <= 1 then
            return(1)
        else
            return(recursive(n - 1) + recursive(n - 1))
    end
```

1.13. Докажите, что следующие утверждения истинны:

а) 17 имеет порядок $O(1)$;

б) $n(n - 1)/2$ имеет порядок $O(n^2)$;

в) $\max(n^3, 10n^2)$ имеет порядок $O(n^3)$;

г) $\sum_{i=1}^{n} i^k$ есть $O(n^{k+1})$ и $\Omega(n^{k+1})$ для целых k;

д) если $p(x)$ — полином степени k с положительным старшим **коэффициентом,** то $p(x)$ есть $O(n^k)$ и $\Omega(n^k)$.

*1.14. Предположим, что $T_1(n)$ есть $\Omega(f(n))$ и $T_2(n)$ — $\Omega(g(n))$. Какие из следующих • утверждений истинны?

а) $T_1(n) + T_2(n)$ есть $\Omega(\max(f(n), g(n)))$;

б) $T_1(n)T_2(n)$ есть $\Omega(f(n)g(n))$.

*1.15. Некоторые авторы определяют нижний порядок роста Ω следующим образом: $f(n)$ есть $\Omega(g(n))$, если существуют такие неотрицательные константы n_0 и с, что для всех л > n_0 выполняется неравенство $f(n) \geq cg(n)$.

а) В рамках этого определения будет ли истинным следующее утверждение: $f(n)$ есть $\Omega(g(n))$ тогда и только тогда, когда $g(n)$ имеет порядок $O(f(n))$?

б) Будет ли утверждение а) истинным для определения Ω из раздела 1.4?

в) Выполните упражнение 1.14 для данного определения Ω.

1.16. Расположите следующие функции в порядке степени роста: а) n, б) $\sqrt{n}$, в) $\log n$, г) $\log \log n$, д) $\log^2 n$, е) $n/\log n$, ж) $\sqrt{n}\log^2$ л, з) $(1/3)^n$, и) $(3/2)^n$, к) 17.

1.17. Предположим, что параметр л приведенной ниже процедуры является положительной целой степенью числа 2, т.е. л = 2, 4, 8, 16, ... Найдите формулу для вычисления значения переменной *count в* зависимости от значения л.

```
procegure mystery ( n: integer );
    var
        x, count: integer;
    begin
        count:= 0;
        x:= 2;
        while x < n do begin
            x:= 2 * x;
            count:= count + 1
```

```
        end;
        writeln(count)
    end
```

1.18. Рассмотрим функцию $max(i, n)$, которая возвращает наибольший из элементов, стоящих в позициях от i до i + n - 1 в целочисленном массиве A. Также предположим, что n является степенью числа 2.

```
procegure max ( i, n: integer ): integer;
    var
        m1, m2: integer;
    begin
        if n = 1 then
            return(A[i])
        else bedin
            m1:= max(i, n div 2);
            m2:= max(i+n div 2, n div 2);
            if ml < m2 then
                return(m2)
            else
                return(m1)
        end
    end
```

а) Пусть $T(n)$ обозначает время выполнения в наихудшем случае программы *max* в зависимости от второго аргумента га. Таким образом, n — число элементов, среди которых ищется наибольший. Запишите рекуррентное соотношение для $T(n)$, в котором должны присутствовать $T(j)$для одного или нескольких значений у, меньших n, и одна или несколько констант, представляющих время выполнения отдельных операторов программы *max*.

б) Запишите в терминах *О*-большое и П верхнюю и нижнюю границы для $T(n)$ в максимально простом виде.

Библиографические замечания

Концепция абстрактных типов данных берет свое начало от типа *class* в языке SIMULA 67 [12]. С тех пор разработано много языков программирования, поддерживающих абстрактные типы данных, среди которых язык Alphard [100], С с классами [105], CLU [69], MESA [42] и Russell [23]. Концепция АТД активно обсуждалась в работах [43] и [122].

[63] — первая большая работа, посвященная систематическому изучению времени выполнения программ. Авторы этой книги в более ранней работе [3] связали временную и емкостную сложности алгоритмов с различными моделями вычислений, таких как машины Тьюринга и машины с произвольным доступом к памяти. Другие библиографические ссылки на работы, посвященные анализу алгоритмов и программ, приведены в библиографических замечаниях к главе 9.

Дополнительный материал о структурном программировании вы найдете в [50], [60], [120] и [125]. Производственные и философские вопросы организации больших программных проектов обсуждаются в работах [15] и [115]. В [61] показано, как создавать полезные программные инструменты для сред программирования.

ГЛАВА 2

Основные абстрактные типы данных

В этой главе мы изучим некоторые из наиболее общих абстрактных типов данных (АТД). Мы рассмотрим списки (последовательности элементов) и два специальных случая списков: стеки, где элементы вставляются и удаляются только на одном конце списка, и очереди, когда элементы добавляются на одном конце списка, а удаляются на другом. Мы также кратко рассмотрим отображения — АТД, которые ведут себя как функции. Для каждого из этих АТД разберем примеры их реализации и сравним по нескольким критериям.

2.1. Абстрактный тип данных „Список"

Списки являются чрезвычайно гибкой структурой, так как их легко сделать большими или меньшими, и их элементы доступны для вставки или удаления в любой позиции списка. Списки также можно объединять или разбивать на меньшие списки. Списки регулярно используются в приложениях, например в программах информационного поиска, трансляторах программных языков или при моделировании различных процессов. Методы управления памятью, которые мы обсудим в главе 12, широко используют технику обработки списков. В этом разделе будут описаны основные **операции**, выполняемые над списками, а далее мы представим структуры данных для списков, которые эффективно поддерживают различные подмножества таких операций.

В математике *список* представляет собой последовательность элементов определенного типа, который в общем случае будем обозначать как **elementtype** (тип элемента). Мы будем часто представлять список в виде последовательности элементов, разделенных запятыми: a_1, a_2, ..., a_n, где $n > 0$ и все a_i имеют тип elementtype. Количество элементов n будем называть *длиной списка.* Если $n > 1$, то a_1 называется *первым элементом,* а a_n — *последним элементом* списка. В случае n = O имеем *пустой список,* который не содержит элементов.

Важное свойство списка заключается в том, что его элементы можно линейно упорядочить в соответствии с их позицией в списке. Мы говорим, что элемент a_i *предшествует* a_{i+1} для i = 1, 2, ..., n — 1 и a_i *следует за* a_{i-1} для i = 2, 3, ..., n. Мы также будем говорить, что элемент a_i *имеет позицию i*. Кроме того, мы постулируем существование позиции, следующей за последним элементом списка. Функция END(L) будет возвращать позицию, следующую за позицией n в n-элементном списке L. Отметим, что позиция END(L), рассматриваемая как расстояние от начала списка, может изменяться при увеличении или уменьшении списка, в то время как другие позиции имеют фиксированное (неизменное) расстояние от начала списка.

Для формирования абстрактного типа данных на основе математического определения списка мы должны задать множество операторов, выполняемых над объекта-

ми типа **LIST**[1] **(Список).** Однако не существует одного множества операторов, выполняемых над списками, удовлетворяющего сразу все возможные приложения. (Это утверждение справедливо и для многих других АТД, рассматриваемых в этой книге.) В этом разделе мы предложим одно множество операторов, а в следующем рассмотрим несколько структур для представления списков и напишем соответствующие процедуры для реализации типовых операторов, выполняемых над списками, в терминах этих структур данных.

Чтобы показать некоторые общие операторы, выполняемые над списками, предположим, что имеем приложение, содержащее список почтовой рассылки, который мы хотим очистить от повторяющихся адресов. Концептуально эта задача решается очень просто: для каждого элемента списка удаляются все последующие элементы, совпадающие с данным. Однако для записи такого алгоритма необходимо определить операторы, которые должны найти первый элемент списка, перейти к следующему элементу, осуществить поиск и удаление элементов.

Теперь перейдем к непосредственному определению множества операторов списка. Примем обозначения: L — список объектов типа elementtype, x — объект этого типа, p — позиция элемента в списке. Отметим, что "позиция" имеет другой тип данных, чья реализация может быть различной для разных реализаций списков. Обычно мы понимаем позиции как множество целых положительных чисел, но на практике могут встретиться другие представления.

1. **INSERT**(x, p, L). Этот оператор вставляет объект x в позицию p в списке L, перемещая элементы от позиции p и далее в следующую, более высокую позицию. Таким образом, если список L состоит из элементов $a_1, a_2, \ldots, a_n$, то после выполнения этого оператора он будет иметь вид $a_1, a_2, \ldots, a_{p-1}, x, a_p, \ldots, a_n$. Если p принимает значение **END**(L), то будем иметь $a_1, a_2, \ldots, a_n, x$. Если в списке L нет позиции p, то результат выполнения этого оператора не определен.
2. **LOCATE**(x, L). Эта функция возвращает позицию объекта x в списке L. Если в списке объект x встречается несколько раз, то возвращается позиция первого от начала списка объекта x. Если объекта x *нет* в списке L, то возвращается **END**(L).
3. **RETRIEVE**(p, L). Эта функция возвращает элемент, который стоит в позиции p в списке L. Результат не определен, если p — **END**(L) или в списке L нет позиции p. Отметим, что элементы должны быть того типа, который в принципе может возвращать функция. Однако на практике мы всегда можем изменить эту функцию так, что она будет возвращать указатель на объект типа elementtype.
4. **DELETE**(p, L). Этот оператор удаляет элемент в позиции p списка L. Так, если список L состоит из элементов $a_1, a_2, \ldots, a_n$, то после выполнения этого оператора он будет иметь вид $a_1, a_2, \ldots, a_{p-1}, a_{p+1}, \ldots, a_n$. Результат не определен, если в списке L нет позиции p или p = **END**(L).
5. **NEXT**(p, L) и **PREVIOUS**(p, L). Эти функции возвращают соответственно следующую и предыдущую позиции от позиции p в списке L. Если p — последняя позиция в списке L, то **NEXT**(p, L) = **END**(L). Функция NEXT не определена, когда p = **END**(L). Функция PREVIOUS не определена, если $p = 1$. Обе функции не определены, если в списке L нет позиции p.
6. **MAKENULL**(L). Эта функция делает список L пустым и возвращает позицию **END**(L).

[1] Строго говоря, тип объекта определен словами "список элементов типа elementtype". Однако мы предполагаем, что реализация списка не зависит от того, что подразумевается под "elementtype", — именно это обстоятельство объясняет то особое внимание, которое мы уделяем концепции списков. Поэтому мы будем использовать термин "тип LIST", а не "список элементов типа elementtype", и в таком же значении будем трактовать другие АТД, зависящие от типов элементов.

7. **FIRST(*L*).** Эта функция возвращает первую позицию в списке *L*. Если список пустой, то возвращается позиция END(*L*).

8. **PRINTLIST(*L*).** Печатает элементы списка *L* в порядке из расположения.

Пример 2.1. Используя описанные выше операторы, создадим процедуру PURGE (Очистка), которая в качестве аргумента использует список и удаляет из него повторяющиеся элементы. Элементы списка имеют тип elementtype, а список таких элементов имеет тип LIST (данного соглашения мы будем придерживаться на протяжении всей этой главы). Определим функцию *same(x, y)*, где *x* и *y* имеют тип elementtype, которая принимает значение true (истина), если *x* и *y* "одинаковые" (same), и значение false (ложь) в противном случае. Понятие "одинаковые", очевидно, требует пояснения. Если тип elementtype, например, совпадает с типом действительных чисел, то мы можем положить, что функция *same(x, y)* будет иметь значение true тогда и только тогда, когда *x* — *y*. Но если тип elementtype является типом записи, содержащей поля почтового индекса (*acctno*), имени (*name*) и адреса абонента (*address*), этот тип можно объявить следующим образом: .

```
type
    elementtype = record
        acctno: integer;
        name: packed array [1..20] of char;
        address: packed array [1..50] of char;
    end
```

Теперь можно задать, чтобы функция *same(x, y)* принимала значение true всякий раз, когда *x.acctno* = *y.acctno*[1].

В листинге 2.1 представлен код процедуры PURGE. Переменные *p* и *q* используются для хранения двух позиций в списке. В процессе выполнения программы слева от позиции *p* все элементы уже не имеют дублирующих своих копий в списке. В каждой итерации цикла (2) - (8) переменная *q* используется для просмотра элементов, находящихся в позициях, следующих за позицией *p*, и удаления дубликатов элемента, располагающегося в позиции p. Затем *p* перемещается в следующую позицию и процесс продолжается.

Листинг 2.1. Программа удаления совпадающих элементов

```
    procedure PURGE ( var L: LIST );
        var
            p, q: position; { p — "текущая" позиция в списке L,
                    q перемещается вперед от позиции p }
        begin
(1)         p:= FIRST(L);
(2)         while p <> END(L) do begin
(3)             q:= NEXT(p, L) ;
(4)             while q <> END(L) do
(5)                 if same(RETRIEVE(p, L), RETRIEVE(q, L)) then
(6)                         DELETE(q, L)
                    else
(7)                     g:= NEXT(q, L) ;
(8)         p:= NEXT(p, L)
            end
        end; { PURGE }
```

В следующем разделе мы покажем объявления типов LIST и position (позиция) и реализацию соответствующих операторов, так что PURGE станет вполне работающей про-

[1] Конечно, если мы собираемся использовать эту функцию для удаления совпадающих записей, то в этом случае необходимо проверить также равенство значений полей имени и адреса.

граммой. Как уже указывалось, программа не зависит от способа представления списков, поэтому мы свободны в экспериментировании с различными реализациями списков.

Необходимо сделать замечание о внутреннем цикле, состоящем из строк (4) - (8) листинга 2.1. При удалении элемента в позиции q, строка (6), элементы в позициях $q + 1$, $q + 2$, ... и т.д. переходят на одну позицию влево. Иногда может случиться, что q является последней позицией в списке, тогда после удаления элемента позиция q становится равной END(L). Если теперь мы выполним оператор в строке (7), NEXT(END(L), L), то получим неопределенный результат. Но такая ситуация невозможна, так как между очередными проверками q <> END(L) в строке (4) может выполниться или строка (6), или строка (7), но не обе сразу.

2.2. Реализация списков

В этом разделе речь пойдет о нескольких структурах данных, которые можно использовать для представления списков. Мы рассмотрим реализации списков с помощью массивов, указателей и курсоров. Каждая их этих реализаций осуществляет определенные операторы, выполняемые над списками, более эффективно, чем другая.

Реализация списков посредством массивов

При реализации списков с помощью массивов элементы списка располагаются в смежных ячейках массива. Это представление позволяет легко просматривать содержимое списка и вставлять новые элементы в его конец. Но вставка нового элемента в середину списка требует перемещения всех последующих элементов на одну позицию к концу массива, чтобы освободить место для нового элемента. Удаление элемента также требует перемещения элементов, чтобы закрыть освободившуюся ячейку.

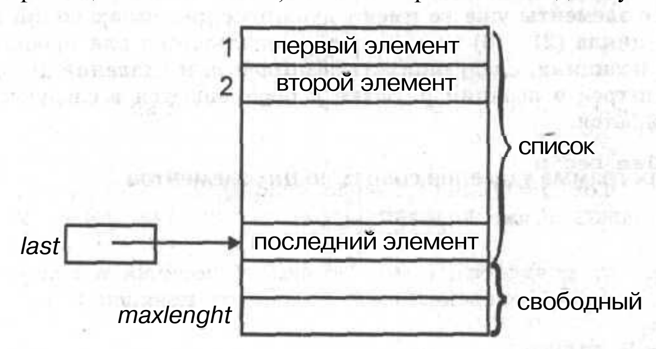


Рис. 2.1. Представление списка с помощью массива (пояснения в тексте)

При использовании массива мы определяем тип LIST как запись, имеющую два поля. Первое поле *elements* (элементы) — это элементы массива, чей размер считается достаточным для хранения списков любой длины, встречающихся в данной реализации или программе. Второе поле целочисленного типа *last* (последний) указывает на позицию последнего элемента списка в массиве. Как видно из рис. 2.1, *i*-й элемент списка, если 1 < i < *last*, находится в i-й ячейке массива. Позиции элементов в списке представимы в виде целых чисел, таким образом, *i*-я позиция — это просто целое число *i*. Функция END(L) возвращает значение *last* + *1*. Приведем необходимые объявления (константа ***maxlength*** определяет максимальный размер массива):

```
const
    maxlength = 100 { или другое подходящее число };
type
```

```
LIST = record
    elements: array[1..maxlength] of elementtype;
    last: integer
end;
position = integer;

function END ( var L: LIST ): position;
begin
    return(L.last + 1)
end; { END }
```

В листинге 2.2 показано, как можно реализовать операторы INSERT, DELETE и LOCATE при представлении списков с помощью массивов. Оператор INSERT перемещает элементы из позиций *p*, *p* + 1, ..., *last* в позиции *p* + 1, *p* + 2, ..., *last* + 1 и помещает новый элемент в позицию *p*. Если в массиве уже нет места для нового элемента, то инициализируется подпрограмма *error* (ошибка), распечатывающая соответствующее сообщение, затем выполнение программы прекращается. Оператор DELETE удаляет элемент в позиции *p*, перемещая элементы из позиций *p* + 1, *p* + 2, ..., *last* в позиции *p*, *p* + 1, ..., *last* — 1. Функция LOCATE последовательно просматривает элементы массива для поиска заданного элемента. Если этот элемент не найден, то возвращается *last* + 1.

Листинг 2.2. Реализация операторов списка

```
procedure INSERT (x: elementtype; p: position; var L: LIST );
  { INSERT вставляет элемент x в позицию p в списке L }
    var
        q: position;
    begin
        if L.last >= maxlength then
            error('Список полон')
        else if (p > L.last + 1) or (p < 1) then
            error('Такой позиции не существует')
        else begin
            for q:= L.last downto p do
                { перемещение элементов из позиций p, p+1, ... на
                    одну позицию к концу списка }
                L.elements[q+1]:= L.elements[q];
            L.last:= L.last + 1;
            L.elements[p]:= x
        end
    end; { INSERT }

procedure DELETE ( p: position; var L: LIST );
  { DELETE удаляет элемент в позиции p списка L }
    var
        q: position;
    begin
        if (p > L.last) or (p < 1) then
            error('Такой позиции не существует')
        else begin
            L.last:= L.last - 1;
            for q:= p to L.last do
                { перемещение элементов из позиций p+1, p+2, ...
                        на одну позицию к началу списка }
                L.elements[q]:= L.elements[q+1]
```

```
        end
    end; { DELETE }

procedure LOCATE ( x: elementtype; L: LIST ): position;
    { LOCATE возвращает позицию элемента x в списке L }
    var
        g: position;
    begin
        for q:= 1 to L.last do
            if L.elements[q] = x then
                return(q);
        return(L.last + 1) { элемент x не найден }
    end; { LOCATE }
```

Легко видеть, как можно записать другие операторы списка, используя данную реализацию списков. Например, функция FIRST всегда возвращает 1, функция NEXT возвращает значение, на единицу большее аргумента, а функция PREVIOUS возвращает значение, на единицу меньшее аргумента. Конечно, последние функции должны делать проверку корректности результата. Оператор **MAKENULL(*L*)** устанавливает *L.last* в 0.

Итак, если выполнению процедуры PURGE (листинг **2.1**) предшествуют определения типа elementtype и функции *same,* объявления типов LIST и position и задание функции END (как показано выше), написание процедуры DELETE (листинг 2.2), подходящая реализация простых процедур FIRST, NEXT и RETRIEVE, то процедура PURGE станет вполне работоспособной программой.

Вначале написание всех этих процедур для управления доступом к основополагающим структурам может показаться трудоемким делом. Но если мы все же подвигнем себя на написание программ в терминах операторов управления абстрактными типами данными, не задерживаясь при этом на частных деталях их реализаций, то сможем изменять программы, изменяя реализацию этих операторов, не выполняя утомительного поиска тех мест в программах, где необходимо внести изменения в форму или способ доступа к основополагающим структурам данных. Эта гибкость может иметь определяющее значение при разработке больших программных проектов. К сожалению, читатель не сможет в полной мере оценить эту сторону использования АТД из-за небольших (по размеру) примеров, иллюстрирующих эту книгу.

Реализация списков с помощью указателей

В этом разделе для реализации однонаправленных списков используются указатели, связывающие последовательные элементы списка. Эта реализация освобождает нас от использования непрерывной области памяти для хранения списка и, следовательно, от необходимости перемещения элементов списка при вставке или удалении элементов. Однако ценой за это удобство становится дополнительная память для хранения указателей.

В этой реализации список состоит из ячеек, каждая из которых содержит элемент списка и указатель на следующую ячейку списка. Если список состоит из элементов $a_1, a_2, \ldots, a_n$, то для $i = 1, 2, \ldots, n-1$ ячейка, содержащая элемент a_i, имеет также указатель на ячейку, содержащую элемент a_{i+1}. Ячейка, содержащая элемент a_n, имеет указатель **nil** (нуль). Имеется также ячейка *header* (заголовок), которая указывает на ячейку, содержащую a_1. Ячейка *header* не содержит элементов списка[1]. В случае пустого списка заголовок имеет указатель **nil,** не указывающий ни на какую ячейку. На рис. 2.2 показан связанный список описанного вида.

---

[1] Объявление ячейки заголовка "полноценной" ячейкой (хотя и не содержащей элементов списка) упрощает реализацию операторов списка на языке Pascal. Можете использовать указатели на заголовки, если хотите каким-либо специальным способом реализовать операторы вставки и удаления в самом начале списка.

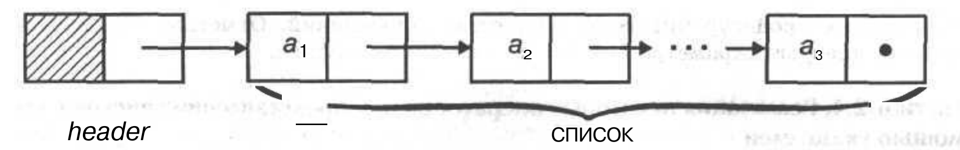


Рис. 2.2. Связанный список

Для однонаправленных списков удобно использовать определение позиций элементов, отличное от того определения позиций, которое применялось в реализации списков с помощью массивов. Здесь для i = 2, 3. . . . *n* позиция *i* определяется как указатель на ячейку, содержащую указатель на элемент a_i. Позиция 1 — это указатель в ячейке заголовка, а позиция **END(*L*)** — указатель в последней ячейке списка *L*.

Бывает, что тип списка совпадает с типом позиций, т.е. является указателем на ячейку, реже на заголовок. Формально определить структуру связанного списка можно следующим образом:

```
type
    celltype = record
        element: elementtype;
        next: t celltype
    end;
    LIST = t celltype;
    position = T celltype;
```

В листинге 2.3 показан код функции **END(*L*).** Результат функции получаем путем перемещения указателя *q* от начала списка к его концу, пока не будет достигнут конец списка, который определяется тем, что *q* становится указателем на ячейку с указателем **nill.** Отметим, что эта реализация функции END неэффективна, так как требует просмотра всего списка при каждом вычислении этой функции. Если необходимо частое использование данной функции, как в программе PURGE (листинг 2.1), то можно сделать на выбор следующее.

1. Применить представление списков, которое не использует указатели на ячейки.
2. Исключить использование функции **END(*L*)** там, где это возможно, заменив ее другими операторами. Например, условие *p* <> **END(*L*)** в строке (2) листинга 2.1 можно заменить условием ***p*↑*.next o*** nill, **но** в этом случае программа становится зависимой от реализации списка.

Листинг 2.3. Функция END

```
     function END .( L: LIST ) : position;
         { END возвращает указатель на последнюю ячейку списка L }
         var
             q: position;
         begin
(1)          g:= L;
(2)          while q↑.next <> nil do
(3)              g:= q↑.next;
(4)          return(q)
         end; { END }
```

Листинг 2.4 содержит процедуры для операторов INSERT, DELETE, LOCATE и **MAKENULL,** которые используются в реализации списков с помощью указателей. Другие необходимые операторы можно реализовать как **одношаговые** процедуры, за исключением PREVIOUS, где требуется просмотр списка с начала. Мы оставляем на-

писание этих процедур читателю в качестве упражнений. Отметим, что многие команды не требуют параметра *L*, списка, поэтому он опущен.

Листинг 2.4. Реализация некоторых операторов при представлении списков с помощью указателей

```
    procedure INSERT ( x: elementtype; p: position );
        var
            temp: position;
        begin
(1)         temp:= p↑.next;
(2)         new(p↑.next) ;
(3)         p↑.next↑.element:= x;
(4)         p↑.next↑.next:= temp
        end; { INSERT }

    procedure DELETE ( p: position );
        begin
            p↑.next:= p↑.next↑.next
        end; { DELETE }

    function LOCATE ( x: elementtype; L: LIST ): position;
        var
            p: position;
        begin
            p:= L;
            while p↑.next o nil do
                if p↑.next↑.element = x then
                    return(p)
                else
                    p:= p↑.next;
            return(p) { элемент не найден }
        end; { LOCATE }

    function MAKENULL ( var L: LIST ): position;
        begin
            new(L);
            L↑.next:= nil;
            return(L)
        end; { MAKENULL }
```

Механизм управления указателями в процедуре INSERT (листинг 2.4) показан на рис. 2.3. Рис. 2.3, а показывает ситуацию перед выполнением процедуры INSERT. Мы хотим вставить новый элемент перед элементом *Ъ,* поэтому задаем *р* как указатель на ячейку, содержащую элемент *Ъ*. В строке (2) листинга создается новая ячейка, а в поле *next* ячейки, содержащей элемент *а,* ставится указатель на новую ячейку. В строке (3) поле *element* вновь созданной ячейки принимает значение *х,* а в строке (4) поле *next* этой ячейки принимает значение переменной *temp,* которая хранит указатель на ячейку, содержащую элемент *Ь*. На рис. 2.3, б представлен результат выполнения процедуры INSERT, где пунктирными линиями показаны новые указатели и номерами (совпадающими с номерами строк в листинге 2.4) помечены этапы из создания.

Процедура DELETE более простая. На рис. 2.4 показана схема манипулирования указателем в этой процедуре. Старые указатели показаны сплошными линиями, а новый — пунктирной.

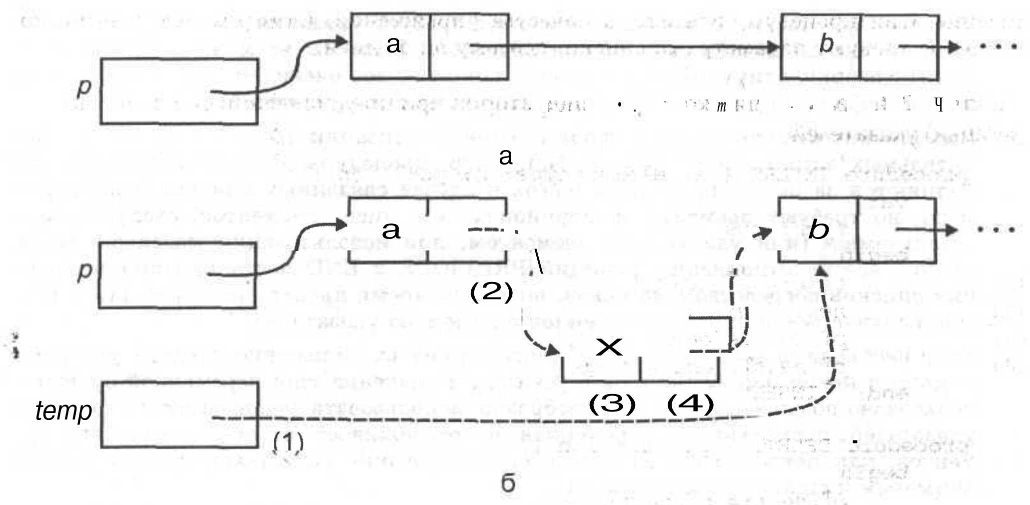


Рис. 2.3. *Диаграммы, иллюстрирующие работу процедуры INSERT*

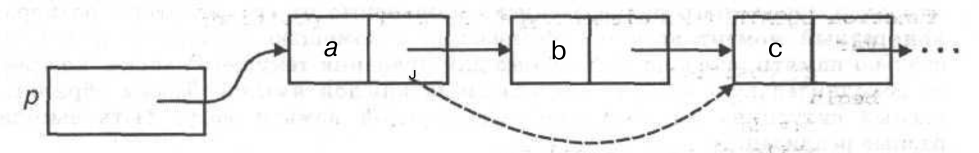


Рис. 2.4. Диаграмма, иллюстрирующая работу процедуры DELETE

Еще раз подчеркнем, что позиции в реализации однонаправленных списков ведут себя не так, как позиции в реализации списков посредством массивов. Предположим, что есть список из трех элементов *a, Ь* и с и переменная *p* типа position (позиция), чье текущее значение равно позиции 3, т.е. это указатель, находящийся в ячейке, содержащей элемент *Ъ,* и указывающий на ячейку, содержащую элемент *c*. Если теперь мы выполним команду вставки нового элемента *x* в позицию 2, так что список примет вид *a, x, Ь, c,* элемент *Ъ* переместится в позицию 3. Если бы мы использовали реализацию списка с помощью массива, то элементы *b* и *c* должны были переместиться к концу массива, так что элемент *b* в самом деле должен оказаться на третьей позиции. Однако при использовании реализаций списков с помощью указателей значение переменной *p* (т.е. указатель в ячейке, содержащей элемент *Ь)* вследствие вставки нового элемента не изменится, продолжая указывать на ячейку, содержащую элемент *c*. Значение этой переменной надо изменить, если мы хотим использовать ее как указатель именно на третью позицию, т.е. как указатель на ячейку, содержащую элемент *b*[1].

Сравнение реализаций

Разумеется, нас не может не интересовать вопрос о том, в каких ситуациях лучше использовать реализацию списков с помощью указателей, а когда — с помощью массивов. Зачастую ответ на этот вопрос зависит от того, какие операторы должны выполняться над списками и как часто они будут использоваться. Иногда аргументом в пользу одной или другой реализации может служить максимальный размер обрабатываемых списков. Приведем несколько принципиальных соображений по этому поводу.

[1] Конечно, можно привести многочисленные примеры ситуаций, когда нужно, чтобы переменная *p* продолжала указывать на позицию элемента *c*.

1. Реализация списков с помощью массивов требует указания максимального размера списка до начала выполнения программ. Если мы не можем заранее ограничить сверху длину обрабатываемых списков, то, очевидно, более рациональным выбором будет реализация списков с помощью указателей.
2. Выполнение некоторых операторов в одной реализации требует больших вычислительных затрат, чем в другой. Например, процедуры INSERT и DELETE выполняются за постоянное число шагов в случае связанных списков любого размера, но требуют времени, пропорционального числу элементов, следующих за вставляемым (или удаляемым) элементом, при использовании массивов. И наоборот, время выполнения функций PREVIOUS и END постоянно при реализации списков посредством массивов, но это же время пропорционально длине списка в случае реализации, построенной с помощью указателей.
3. Если необходимо вставлять или удалять элементы, положение которых указано с помощью некой переменной типа position, и значение этой переменной будет использовано **позднее,** то не целесообразно использовать реализацию с помощью указателей, поскольку эта переменная не "отслеживает" вставку и удаление элементов, как показано выше. Вообще использование указателей требует особого внимания и тщательности в работе.
4. Реализация списков с помощью массивов расточительна в отношении компьютерной памяти, поскольку резервируется объем памяти, достаточный для максимально возможного размера списка независимо от его реального размера в конкретный момент времени. Реализация с помощью указателей использует столько памяти, сколько необходимо для хранения текущего списка, но требует дополнительную память для указателя каждой ячейки. Таким образом, в разных ситуациях по критерию используемой памяти могут быть выгодны разные реализации.

Реализация списков на основе курсоров

Некоторые языки программирования, например Fortran и Algol, не имеют указателей. Если мы работаем с такими языками, то можно смоделировать указатели с помощью курсоров, т.е. целых чисел, которые указывают на позиции элементов в массивах. Для всех списков элементов, имеющих тип elementtype, создадим один массив *SPACE* (область данных), состоящий из записей. Каждая запись будет состоять из поля *element* для элементов списка и поля *next* для целых чисел, используемых в качестве курсора. Чтобы **создать** описанное представление, определим

```
var
    SPACE: array [1..maxlength] of record
        element: elementtype;
        next: integer
    end
```

Для списка *L* объявим целочисленную переменную (например, *Lhead) в* качестве заголовка списка *L.* Можно трактовать *Lhead* как курсор ячейки заголовка массива *SPACE с* пустым значением поля *element.* Операторы списка можно реализовать точно так же, как описано выше в случае использования указателей.

Здесь мы рассмотрим другую реализацию, позволяющую использовать ячейки заголовков для специальных случаев вставки и удаления элементов в позиции 1. (Эту же технику можно использовать и в однонаправленных списках.) Для списка *L* значение ***SPACE*[*Lhead*].*element***равно первому элементу списка, значение ***SPACE*[*Lhead*].*next***является индексом ячейки массива, содержащей второй элемент списка, и т.д. Нулевое значение *Lhead* или нуль в поле *next* указывает на "указатель nil", это означает, что последующих элементов нет.

Список будет иметь целочисленный тип, поскольку заголовок, представляющий список в целом, является целочисленной переменной. Позиции также имеют тип целых чисел. Мы изменим соглашение о том, что позиция *i* в списке *L* является индексом ячейки, содержащей (i - 1)-й элемент списка *L*, поскольку поле *next* этой ячейки содержит курсор, указывающий на *i*-й элемент списка. При необходимости первую позицию любого списка можно представить посредством О. Поскольку имя списка является обязательным параметром операторов, использующих позиции элементов, с помощью имен можно различать первые позиции различных списков. Позиция END(*L*) — это индекс последнего элемента списка *L*.

На рис. 2.5 показаны два списка, *L* — *a*, *b*, *c* и *M* = *d*, *e*, вложенные в один массив *SPACE* длиной 10. Отметим, что все ячейки массива, незанятые элементами списков, образуют отдельный список, называемый *свободным* (ниже в листингах он обозначается *available)*. Этот список используется как "источник" пустых ячеек при вставке элементов в любой список, а также как место хранения перед дальнейшим использованием освободившихся (после удаления элементов) ячеек.

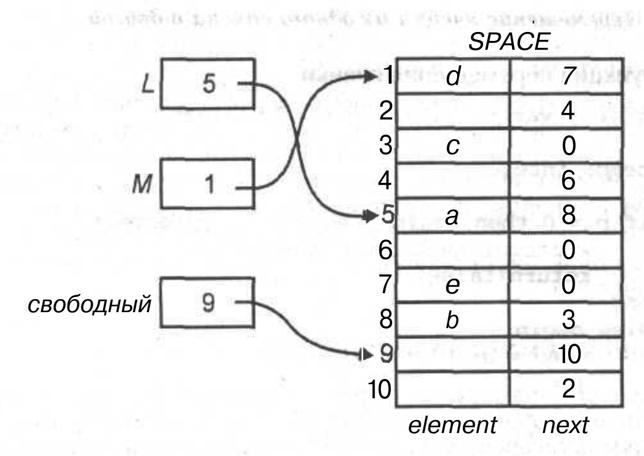


Рис. 2.5. Реализация связанных списков с использованием курсоров

Для вставки элемента *x* в список L мы берем первую ячейку из свободного списка и помещаем ее в нужную позицию списка *L*. Далее элемент *x* помещается в поле *element* этой ячейки. Для удаления элемента *x* из списка L мы удаляем ячейку, содержащую элемент *x*, из списка и помещаем ее в начало свободного списка. Эти два действия являются частными случаями следующей операции. Пусть есть ячейка С, на которую указывает курсор *p*, необходимо сделать так, чтобы на ячейку С указывал другой курсор *q*, курсор ячейки С должен указывать на ячейку, на которую ранее указывал курсор *q*, курсор *p* должен указывать на ячейку, на которую до выполнения этой операции указывал курсор ячейки С. Другими словами, надо переместить ячейку из одного места списка (указываемого курсором *p)* в другое место того же или другого списка, новое место указывается курсором *q*. Например, если необходимо удалить элемент *b* из списка *L* (рис. 2.5), то С — это строка 8 в массиве *SPACE*, *p* равно *SPACE*[5].*next*,а курсор *q* указывает на первую ячейку свободного списка. На рис. 2.6 схематически показан процесс перемещения ячейки С из одного списка в другой (курсоры до и после выполнения операции показаны соответственно сплошными и пунктирными линиями). Код функции *move* (перемещение), выполняющей описанную операцию, приведен в листинге 2.5. Если в массиве нет ячейки С, то функция возвращает значение false (ложь), при успешном выполнении функции возвращается значение true (истина).

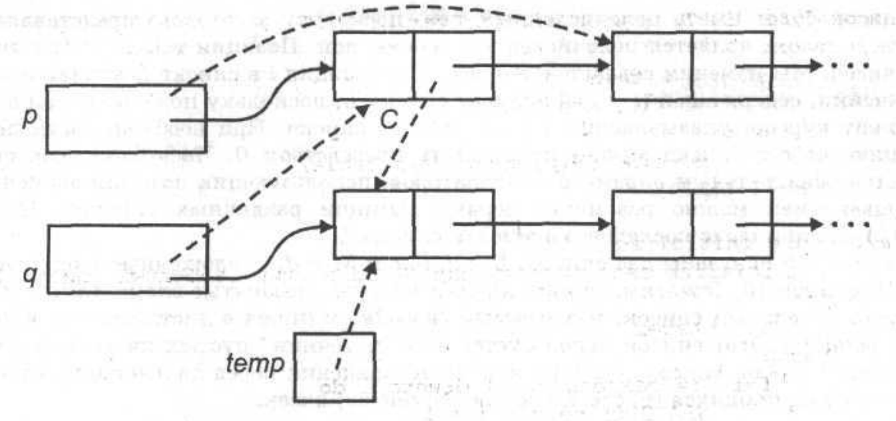


Рис. 2.6. Перемещение ячейки из одного списка в другой

Листинг 2.5. Функция перемещения ячейки

```
function move ( var p, g: integer ): boolean;
    var
        temp: integer;
    begin
        if p = 0 then begin { ячейка не существует }
            writeln('Ячейка не существует')
            return(false)
        end
        else begin
            temp:= q;
            q:= p;
            p:=  SPACE[q].next;
            SPACE[q].next:= temp;
            return(true)
        end
    end; { move }
```

В **листинге 2.6** приведены коды процедур INSERT и DELETE, а также процедуры *initialize* (инициализация), связывающей ячейки массива *SPACE* в свободный список. В этих процедурах опущены проверки "нештатных" ситуаций, которые могут вызвать ошибки (оставляем написание кода этих проверок читателю в качестве упражнений). Кроме того, в Качестве упражнений оставляем читателю реализацию других операторов, выполняемых над списками (их код подобен коду аналогичных процедур при использовании указателей).

Листинг 2.6. Реализация некоторых операторов списка при использовании курсоров

```
procedure INSERT ( x: elementtype; p: position; var L: LIST );
    begin
        if p = 0 then begin { вставка x в первую позицию }
            if move(available, L) then
                SPACE[L].element:= x
        end
        else { вставка x в позицию, отличную от первой }
            if move(available, SPACE[p].next) then
                SPACE[SPACE[p].next].element:= x
    end; { INSERT }
```

```
procedure DELETE ( p: position; var L: LIST );
    begin
        if p = 0 then
            move(L, available)
        else
            move(SPACE[p].next, available)
    end; { DELETE }

procedure initialize
    { initialize связывает ячейки SPACE в один свободный список }
    var
        i: integer
    begin
        for i:= maxlength - 1 downto 1 do
            SPACE[i].next:= i + 1;
        available:= 1;
        SPACE[maxlength] .next:= 0
            { помечен конец свободного списка }
    end; { initialize }
```

Дважды связные списки

Во многих приложениях возникает необходимость организовать эффективное перемещение по списку как в прямом, так и в обратном направлениях. Или по заданному элементу нужно быстро найти предшествующий ему и последующий элементы. В этих ситуациях можно дать каждой ячейке указатели и на следующую, и на предыдущую ячейки списка, т.е. организовать *дважды связный список* (рис. 2.7). В главе 12 будут приведены ситуации, когда использование дважды связных списков особенноэффективно.

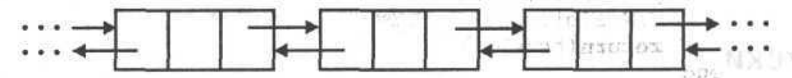


Рис. 2.7. Дважды связный список

Другое важное преимущество дважды связных списков заключается в том, что мы можем использовать указатель ячейки, содержащей *i*-й элемент, для определения *i*-й позиций — вместо использования указателя предшествующей ячейки. Но ценой этой возможности являются дополнительные указатели в каждой ячейке и определенное удлинение некоторых процедур, реализующих основные операторы списка. Если мы используем указатели (а не курсоры), то объявление ячеек, содержащих элементы списка и два указателя, можно выполнить следующим образом (*previous* — поле, содержащее указатель на предшествующую ячейку):

```
type
    celltype = record
        element: elementtype;
        next, previous: t celltype
    end;
    position = t celltype;
```

Процедура удаления элемента в позиции p дважды связного списка показана в листинге 2.7. На рис. 2.8 Приведена схема удаления элемента в предположении, что

удаляемая ячейка не является ни первой, ни последней в списке[1]. На этой схеме сплошными линиями показаны указатели до удаления, а пунктирными — после удаления элемента. В процедуре удаления сначала с помощью указателя поля *previous* определяется положение предыдущей ячейки. Затем в поле *next* этой (предыдущей) ячейки устанавливается указатель, указывающий на ячейку, следующую за позицией *p*. Далее подобным образом определяется следующая за позицией *p* ячейка и в ее поле *previous* устанавливается указатель на ячейку, предшествующую позиции *p*. Таким образом, ячейка в позиции *p* исключается из цепочек указателей и при необходимости может быть использована повторно.

Листинг 2.7. Удаление элемента из дважды связного списка

```
procedure DELETE ( var p: position );
    begin
        if p↑.previous <> nil then
            { удаление ячейки, которая не является первой }
            p↑.previous↑.next:= p↑.next;
        if p↑.next <> nil then
            { удаление ячейки, которая не является последней }
            p↑.next↑.previous:= p↑.previous
    end; { DELETE }
```

p

Рис. 2.8. Схема удаления ячейки в дважды связном списке

2.3. Стеки

Стек — это специальный тип списка, в котором все вставки и удаления выполняются только на одном конце, называемом *вершиной* (top). Стеки также иногда называют "магазинами", а в англоязычной литературе для обозначения стеков еще используется аббревиатура LIFO (last-in-first-out — последний вошел — первый вышел). Интуитивными моделями стека могут служить колода карт на столе при игре в покер, книги, сложенные в стопку, или стопка тарелок на полке буфета; во всех этих моделях взять можно только верхний предмет, а добавить новый объект можно, только положив его на верхний. Абстрактные типы данных семейства STACK (Стек) обычно используют следующие пять операторов.

1. MAKENULL(*S*). Делает стек S пустым.
2. TOP(S). Возвращает элемент из вершины стека S. Обычно вершина стека идентифицируется позицией 1, тогда TOP(*S*) можно записать в терминах общих операторов списка как RETRIEVE(FIRST(S), S).
3. POP(S). Удаляет элемент из вершины стека (выталкивает из стека), в терминах операторов списка этот оператор можно записать как DELETE(FIRST(S), S).

[1] В этой связи отметим, что на практике обычно делают так, что ячейка заголовка дважды связного списка "замыкает круг" ячеек, т.е. указатель поля *previous* ячейки заголовка указывает на последнюю ячейку, а указатель поля *next* — на первую. Поэтому при такой реализации дважды связного списка нет необходимости в выполнении проверки на "нулевой указатель".

Иногда этот оператор реализуется в виде функции, возвращающей удаляемый элемент.

4. PUSH(x, S). Вставляет элемент x в вершину стека S (заталкивает элемент в стек). Элемент, ранее находившийся в вершине стека, становится элементом, следующим за вершиной, и т.д. В терминах общих операторов списка данный оператор можно записать как INSERT(x, FIRST(S), S).

5. EMPTY(S). Эта функция возвращает значение true (истина), если стек S пустой, и значение false (ложь) в противном случае.

Пример 2.2. Все текстовые редакторы имеют определенные символы, которые служат в качестве *стирающих символов* (erase character), т.е. таких, которые удаляют (стирают) символы, стоящие перед ними; эти символы "работают" как клавиша <Backspace> на клавиатуре компьютера. Например, если символ # определен стирающим символом, то строка *abc#d##e* в действительности является строкой *ae*. Здесь первый символ # удаляет букву *c*, второй стирающий символ стирает букву *d*, а третий — букву *Ъ*.

Текстовые редакторы имеют также *символ-убийцу* (kill character), который удаляет все символы текущей строки, находящиеся перед ним. В этом примере в качестве символа-убийцы определим символ @.

Операции над текстовыми строками часто выполняются с использованием стеков. Текстовый редактор поочередно считывает символы, если считанный символ не является ни символом-убийцей, ни стирающим символом, то он помещается в стек. Если вновь считанный символ — стирающий символ, то удаляется символ в вершине стека. В случае, когда считанный символ является символом-убийцей, редактор очищает весь стек. В листинге 2.7 представлена программа, выполняющая описанные действия.[1]

Листинг 2.7. Программа, реализующая действия стирающего символа и символа-убийцы

```
procedure EDIT;
    var
        s: STACK;
        c: char;
    begin
        MAKENULL(S);
        while not eoln do begin
            read(c);
            if c = '#' then
                POP(S)
            else if c = '@' then
                MAKENULL(S);
            else { c — обычный символ }
                PURSH(c, S)
        end;
        печать содержимого стека S в обратном порядке
    end; { EDIT }
```

В этой программе тип STACK можно объявить как список символов. Процесс вывода содержимого стека в обратном порядке в последней строке программы требует небольшой хитрости. Выталкивание элементов из стека по одному за один раз в принципе позволяет получить последовательность элементов стека в обратном порядке. Некоторые реализации стеков, например с помощью массивов, как описано ни-

[1] **В этом листинге используется стандартная функция языка Pascal *eoln*, возвращающая значение true, если текущий символ — символ конца строки. — *Прим. ред.***

же, позволяют написать простые процедуры для печати содержимого стека, начиная с обратного конца стека. В общем случае необходимо извлекать элементы стека по одному и вставлять их последовательно в другой стек, затем распечатать элементы из второго стека в прямом порядке. □

Реализация стеков с помощью массивов

Каждую реализацию списков можно рассматривать как реализацию стеков, поскольку стеки с их операторами являются частными случаями списков с операторами, выполняемыми над списками. Надо просто представить стек в виде однонаправленного списка, так как в этом случае операторы PURCH и POP будут работать только с ячейкой заголовка и первой ячейкой списка. Фактически заголовок может быть или указателем, или курсором, а не полноценной ячейкой, поскольку стеки не используют такого понятия, как "позиция", и, следовательно, нет необходимости определять позицию 1 таким же образом, как и другие позиции.

Однако реализация списков на основе массивов, описанная в разделе 2.2, не очень подходит для представления стеков, так как каждое выполнение операторов PURCH и POP в этом случае требует перемещения всех элементов стека и поэтому время их выполнения пропорционально числу элементов в стеке. Можно более рационально приспособить массивы для реализации стеков, если принять во внимание тот факт, что вставка и удаление элементов стека происходит только через вершину стека. Можно зафиксировать "дно" стека в самом низу массива (в ячейке с наибольшим индексом) и позволить стеку расти вверх массива (к ячейке с наименьшим индексом). Курсор с именем *top* (вершина) будет указывать положение текущей позиции первого элемента стека. Схематично такое представление стека показано на рис. 2.9.

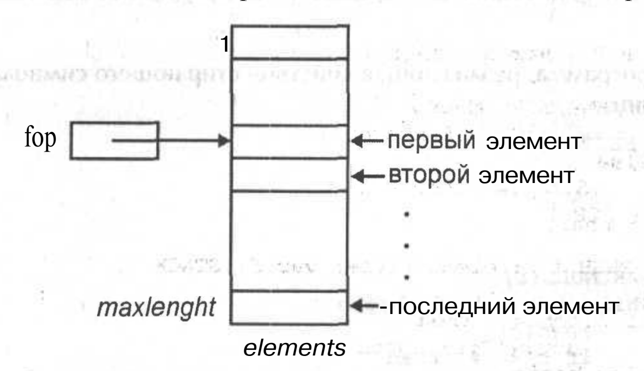


Рис. 2.9. Реализация стека на основе массива

Для такой реализации стеков можно определить абстрактный тип STACK следующим образом:

```
type
    STACK = record
        top: integer;
        element: array[1..maxlength] of elementtype
    end;
```

В этой реализации стек состоит из последовательности элементов ***element*[*top*]**, ***element*[*top*+ 1]**, ..., ***element*[*maxlength*]**. Отметим, если *top* = *maxlength* + 1, то стек пустой.

Процедуры для реализаций пяти типовых операторов, выполняемых над стеками, представлены в листинге 2.9. Заметим, что значение, возвращаемое функцией TOP, имеет тип elementtype, который должен быть разрешенным типом результата, воз-

вращаемого функцией. Если это не так, данную функцию нужно преобразовать в процедуру или она должна возвращать указатель на элемент типа elementtype.

Листинг 2.9. Реализация операторов, выполняемых над стеками

```
procedure MAKENULL ( var S: STACK );
    begin
        S.top:= maxlength + 1
    end; { MAKENULL }

function EMPTY ( S: STACK ): boolean;
    begin
        if S.top > maxlength then
            return(true)
        else
            return(false)
    end; { EMPTY }

function TOP ( var S: STACK ): elementtype;
    begin
        if EMPTY(S) then
            error('Стек пустой')
        else
            return(S.elements[S.top])
    end; { TOP }

procedure POP ( var S: STACK );
    begin
        if EMPTY(S) then
            error('Стек пустой')
        else
            S.top:= S.top + 1
    end; { POP }

procedure PUSH ( x: elementtype; var S: STACK );
    begin
        if S.top = 1 then
            error('Стек полон')
        else begin
            S.top:= S.top - 1
            S.elements[S.top]:= x
        end
    end; { PUSH }
```

2.4. Очереди

Другой специальный тип списка — *очередь* (queue), где элементы вставляются с одного конца, называемого *задним* (rear), а удаляются с другого, *переднего* (front). Очереди также называют "списками типа FIFO" (аббревиатура FIFO расшифровывается как first-in-first-out: первым вошел — первым вышел). Операторы, выполняемые над очередями, аналогичны операторам стеков. Существенное отличие между ними состоит в том, что вставка новых элементов осуществляется в конец списка, а не в начало, как в стеках. Кроме того, различна устоявшаяся терминология для стеков и очередей. Мы будем использовать следующие операторы для работы с очередями.

1. **MAKENULL(*Q*)** очищает очередь ***Q***, делая ее пустой.
2. **FRONT(*Q*)** — функция, возвращающая первый элемент очереди ***Q***. Можно реализовать эту функцию с помощью операторов списка как **RETRIEVE(FIRST(*Q*), *Q*)**.
3. **ENQUEUE(*x*, *Q*)** вставляет элемент *x* в конец очереди ***Q***. С помощью операторов списка этот оператор можно выполнить следующим образом: **INSERT(*x*, END(*Q*), *Q*)**.
4. **DEQUEUE(*Q*)** удаляет первый элемент очереди ***Q***. Также реализуем с помощью операторов списка как **DELETE(FIRST(*Q*), *Q*)**.
5. **EMPTY(*Q*)** возвращает значение true тогда и только тогда, когда *Q* является пустой очередью.

Реализация очередей с помощью указателей

Как и для стеков, любая реализация списков допустима для представления очередей. Однако учитывая особенность очереди (вставка новых элементов только с одного, заднего, конца), можно реализовать оператор ENQUEUE более эффективно, чем при обычном представлении списков. Вместо перемещения списка от начала к концу каждый раз при пополнении очереди мы можем хранить указатель (или курсор) на последний элемент очереди. Как и в случае со стеками, можно хранить указатель на начало списка — для очередей этот указатель будет полезен при выполнении команд FRONT и DEQUEUE. В языке Pascal в качестве заголовка можно использовать динамическую переменную и поместить в нее указатель на начало очереди. Это позволяет удобно организовать очищение очереди.

Сейчас мы рассмотрим реализацию очередей с использованием указателей языка Pascal. Читатель может разработать аналогичную реализацию с применением курсоров, но в случае очередей мы имеем возможность более эффективного их (очередей) представления, основанного на массивах, и с непосредственным использованием указателей, чем при попытке моделирования указателей с помощью курсоров. В конце этого раздела мы обсудим также реализацию очередей на основе так называемых "циклических массивов". Для начала реализации очередей с помощью массивов необходимо сделать объявление ячеек следующим образом:

```
type
    celltype = record
        element: elementtype;
        next: T celltype
    end;
```

Теперь можно определить список, содержащий указатели на начало и конец очереди. Первой ячейкой очереди является ячейка заголовка, в которой поле *element* игнорируется. Это позволяет, как указывалось выше, упростить представление для любой очереди. Мы определяем **АТД** QUEUE (Очередь)

```
type
    QUEUE = record
        front, rear: T celltype
    end;
```

В листинге 2.10 представлены программы для пяти операторов, выполняемых над очередями. В процедуре **MAKENULL** первый оператор *new(**Q.front**)*определяет динамическую переменную (ячейку) типа celltype и назначает ей адрес ***Q.front***. Второй оператор этой процедуры задает значение поля *next* этой ячейки как nil. Третий оператор делает заголовок для первой и последней ячеек очереди.

Процедура **DEQUEUE(*Q*)** удаляет первый элемент из очереди ***Q***, "отсоединяя" старый заголовок от очереди. Первым элементом списка становится новая динамическая переменная ячейки заголовка.

Листинг 2.10. Реализация операторов очередей

```
procedure MAKENULL  (var Q: QUEUE );
    begin
        new(Q.front); { создание ячейки заголовка }
        Q.front↑.next:= nil;
        Q.rear:= Q.front
    end; { MAKENULL }

function EMPTY  ( Q: QUEUE ): boolean;
    begin
        if Q.front = Q.rear then
            return(true)
        else
            return(true)
    end; { EMPTY }

function FRONT  ( Q: QUEUE ): elementtype;
    begin
        if EMPTY(Q) then
            error('Очередь пуста')
        else
            return(Q.front↑.next↑.element)
    end; { FRONT }

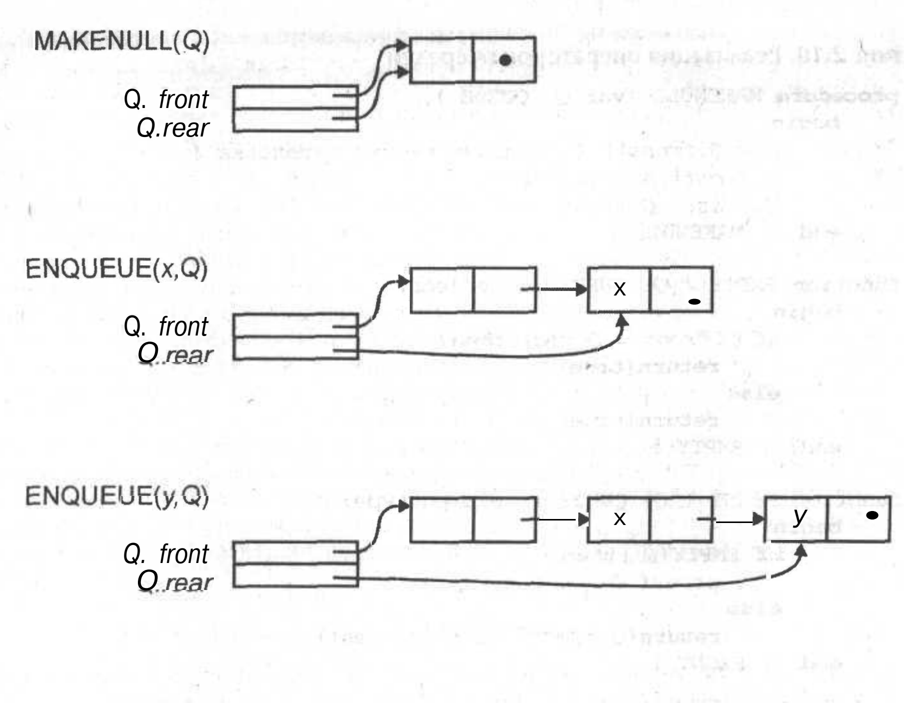
procedure ENQUEUE  ( x: elementtype; var Q: QUEUE );
    begin
        new(Q.rear↑.next);
        Q.rear:= Q.rear↑.next;
        Q.rear↑.element:= x;
        Q.rear↑.next:= nil
    end; { ENQUEUE }

procedure DEQUEUE  ( var Q: QUEUE );
    begin
        if EMPTY(Q) then
            error('Очередь пуста')
        else
            Q.front:= Q.front↑.next
    end; { DEQUEUE }
```

На рис. 2.10 показан результат последовательного применения команд **MAKENULL(*Q*), ENQUEUE(*x*,** Q), **ENQUEUE(*y*,** Q) и **DEQUEUE(*Q*).** Отметим, что после исключения из очереди оператором **DEQUEUE(*Q*)** элемента x он остается в поле *element* ячейки заголовка, но перестает быть частью очереди.

Реализация очередей с помощью циклических массивов

Реализацию списков посредством массивов, которая рассматривалась в разделе 2.2, можно применить для очередей, но в данном случае это не рационально. Действительно, с помощью указателя на последний элемент очереди можно выполнить оператор DEQUEUE за фиксированное число шагов (независимое от длины очереди), но оператор ENQUEUE, который удаляет первый элемент, требует перемещения всех элементов очереди на одну позицию в массиве. Таким образом, ENQUEUE имеет время выполнения $\Omega(n)$, где n — длина очереди.



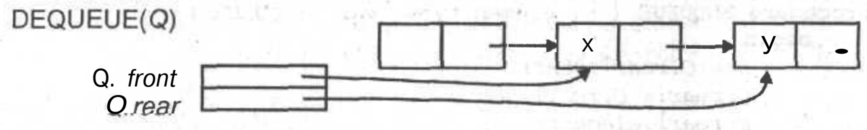


Рис. *2.10. Выполнение последовательности операторов*

Чтобы избежать этих вычислительных затрат, воспользуемся другим подходом. Представим массив в виде циклической структуры, где первая ячейка массива следует за последней, как показано на рис. 2.11. Элементы очереди располагаются в "круге" ячеек в последовательных позициях[1], конец очереди находится по часовой стрелке на определенном расстоянии от начала. Теперь для вставки нового элемента в очередь достаточно переместить указатель *Q.rear* (указатель на конец очереди) на одну позицию по часовой стрелке и записать элемент в эту позицию. При удалении элемента из очереди надо просто переместить указатель *Q.front* (указатель на начало очереди) по часовой стрелке на одну позицию. Отметим, что при таком представлении очереди операторы ENQUEUE и DEQUEUE выполняются за фиксированное время, независимое от длины очереди.

Есть одна сложность представления очередей с помощью циклических массивов и в любых вариациях этого представления (например, когда указатель *Q.rear* указывает по часовой стрелке на позицию, следующую за последним элементом, а не на сам последний элемент). Проблема заключается в том, что только по формальному признаку взаимного расположения указателей *Q.rear* и *Q.front* нельзя сказать, когда очередь пуста, а когда заполнила весь массив. (Конечно, можно ввести специальную переменную, которая будет принимать значение true тогда и только тогда, когда очередь пуста, но если мы не собираемся вводить такую переменную, то необходимо предусмотреть иные средства, предотвращающие переполнение массива.)

---

[1] Отметим, что "последовательность" (непрерывность) позиций здесь понимается как "циклическая непрерывность". Например, очередь из четырех элементов может последовательно занимать две последние и две первые ячейки массива.

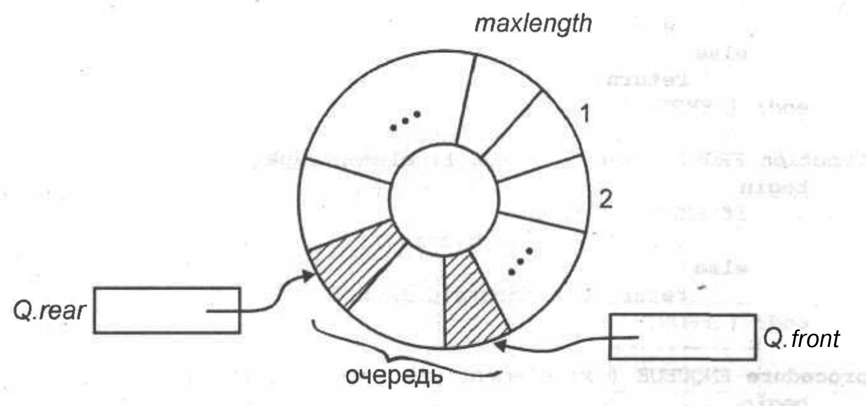


Рис. 2.11. Реализация очереди посредством циклического массива

Чтобы разобраться в этой проблеме, предположим, что очередь состоит из *maxlength* элементов (рис. 2.11), т.е. полностью заполнила массив. Тогда указатель *Q.rear* указывает на позицию рядом с ***Q.front***, но находящуюся против часовой стрелки. Что будет, если очередь пуста? Чтобы увидеть представление пустой очереди, сначала положим, что очередь состоит из одного элемента. Тогда указатели *Q.rear* и *Q.front* указывают на одну и ту же позицию. Если мы удалим из очереди этот элемент, то указатель *Q.front* переместится на одну позицию по часовой стрелке, оформляя пустую очередь. Таким образом, в случае пустой очереди указатель *Q.rear* указывает на позицию рядом с *Q.front,* находящуюся против часовой стрелки, т.е. точно так же, как и при полном заполнении массива. Отсюда следует вывод, что без введения каких-либо механизмов определения пустых очередей при максимальной длине массива *maxlength* мы не можем позволить очереди иметь более *maxlength* - 1 элементов.

Теперь представим пять команд, выполняемых над очередями, использующими описанную реализацию. Формально очереди здесь определяются следующим образом:

```
type
    QUEUE = record
        elements: array[1..maxlength] of elementtype;
        front, rear: integer
    end;
```

Соответствующие процедуры показаны в листинге **2.11.** Функция ***addone(i)*** добавляет единицу к позиции *i* в "циклическом" смысле.

Листинг 2.11. Реализация очередей с помощью циклических массивов

```
function addone ( i: integer ): integer;
    begin
        return((i mod maxlength) + 1)
    end; { addone }

procedure MAKENULL ( var Q: QUEUE );
    begin
        Q.front:= 1;
        Q.rear:= maxlength
    end; { MAKENULL }

function EMPTY ( var Q: QUEUE ): boolean;
    begin
        if addone(Q.rear) = Q.front then
```

```
            return(true)
        else
            return(false)
    end; { EMPTY }

function FRONT ( var Q: QUEUE ): elementtype;
    begin
        if EMPTY(Q) then
            error('Очередь пустая')
        else
            return(Q.elements[Q.front])
    end; { FRONT }

procedure ENQUEUE ( x: elementtype; var Q: QUEUE );
    begin
        if addone(addone(Q.rear)) = Q.front then
            error('Очередь полная')
        else begin
            Q.rear:= addone(Q.rear);
            Q.elements[Q.rear]:= x
        end
    end; { ENQUEUE }

procedure DEQUEUE( var Q: QUEUE );
    begin
        if EMPTY(Q) then
            error('Очередь пустая')
        else
            Q.front:= addone(Q.front)
    end; { DEQUEUE }
```

2.5. Отображения

Отображение — это функция, определенная на множестве элементов (области определения) одного типа (будем обозначать его **domaintype** — тип области определения функции) и принимающая значения из множества элементов (области значений) другого типа, этот тип обозначим rangetype — тип области значений (конечно, типы domaintype и rangetype могут совпадать). Тот факт, что отображение M ставит в соответствие элемент d типа domaintype из области определения элементу r типа rangetype из области значений, будем записывать как $M(d) = r$.

Некоторые отображения, подобные $square(i) = i^2$, легко реализовать с помощью функций и арифметических выражений языка Pascal. Но для многих отображений нет очевидных способов реализации, кроме хранения для каждого d значения $M(d)$. Например, для реализации функции, ставящей в соответствие работникам их недельную зарплату, требуется хранить текущий заработок каждого работника. В конце этого раздела мы опишем методы реализации таких функций.

Рассмотрим операторы, которые можно выполнить над отображением M. Например, по заданному элементу d типа domaintype мы хотим получить $M(d)$ или узнать, определено ли $M(d)$ (т.е. узнать, принадлежит ли элемент d области определения M). Или хотим добавить новые элементы в текущую область определения M и поставить им в соответствие элементы из области значений. Очевидна также необходимость иметь оператор, который изменял бы значение $M(d)$. Кроме того, нужно средство "обнуления" отображения, переводящее любое отображение в *пустое отображение,*

т.е. такое, у которого область определения пуста. Наши пожелания можно обобщить в следующие три команды.

1. MAKENULL(*M*). Делает отображение *M* пустым.
2. ASSIGN(*M, d, r*). Делает *M(d)* равным *r* независимо от того, как *M(d)* было определено ранее.
3. COMPUTE(*M, d, r*). Возвращает значение true и присваивает переменной *r* значение *M(d),* если последнее определено, и возвращает false в противном случае.

Реализация отображений посредством массивов

Во многих случаях тип элементов области определения отображения является простым типом, который можно использовать как тип индексов массивов. В языке Pascal типы индексов включают все конечные интервалы целых чисел, например 1..100 или 17..23, строковый тип, диапазоны символов, подобные 'A'...'Z',и нечисловые типы, например *север, восток, юг, запад.* В частности, в программах кодирования можно применить отображение *crypt* (шифратор) с множеством 'A'...'Z'и в качестве области определения, и в качестве области значений, так что *crypt*(*текст*)будет кодом текста *текст.*

Такие отображения просто реализовать с помощью массивов, предполагая, что некоторые значения типа rangetype могут иметь статус "неопределен". Например, для отображения *crypt,* описанного выше, область значений можно определить иначе, чем 'A'...'Z',и использовать символ '?' для обозначения "неопределен".

Предположим, что элементы области определения и области значений имеют соответственно типы domaintype и rangetype и что тип domaintype является базовым типом языка Pascal. Тогда тип MAPPING (Отображение) можно объявить следующим образом:

```
type
    MAPPING = array[domaintype] of rangetype;
```

Предполагая, что "неопределен" — это константа типа rangetype и что константы firstvalue и lastvalue равны первому и последнему элементу области определения[1], можно реализовать три перечисленных выше оператора, выполняемых над отображениями, как показано в листинге 2.12.

Листинг 2.12. Реализация операторов отображений посредством массива

```
procedure MAKENULL ( var M: MAPPING ) ;
    var
        i: domaintype;
    begin
        for i:= firstvalue to lastvalue do
            M[i]:= неопределен
    end; { MAKENULL }

procedure ASSIGN ( var M: MAPPING; d: domaintype; r: rangetype);
    begin
        M[d] := r
    end; { ASSIGN }

function COMPUTE ( var M: MAPPING;
        d: domaintype; r: rangetype): boolean;
    begin
```

[1] **Например, firstvalue = 'A'и lastvalue = 'Z', если область определения совпадает с множеством 'A'..'Z'.**

```
        if M[d]:= неопределен then
            return(false)
        else begin
            r:= M[d] ;
            return(true)
        end
    end; { COMPUTE }
```

Реализация отображений посредством списков

Существует много реализаций отображений с конечной областью **определения.** Например, во многих ситуациях отличным выбором будут хэш-таблицы, которые мы рассмотрим в главе 4. **Другие** отображения с конечной областью определения можно представить в виде списка пар (d_1, r_1), (d_2, r_2), ..., (d_k, r_k), где d_1, d_2 . . . d» — все текущие элементы области определения, а r_1, r_2, ..., r_k — значения, ассоциированные с rf; (i = 1, 2, ..., k). Далее можно использовать любую реализацию списков.

Абстрактный тип данных MAPPING можно реализовать как список типа **elementtype**, если сделано объявление

```
  type
      elementtype = record
          domain: domaintype
          range: rangetype
      end;
```

и затем объявлен тип MAPPING так, как мы ранее объявляли тип LIST (элементов типа elementtype), и в соответствии с той реализацией списков, которую мы выбрали. В листинге 2.13 приведены листинги трех операторов отображения, записанные через операторы списков типа LIST.

Листинг 2.13. Реализация отображения посредством списков

```
    procedure MAKENULL ( var M: MAPPING );
        { точно такая же, как и для списков }

    procedure ASSING ( var M: MAPPING; d: domaintype; r: rangetype );
        var
            x: elementtype; { пара (d, r) }
            p: position; { используется для перехода по списку M }
        begin
            x.domain:= d;
            x.range:= r;
            p:= FIRST(M) ;
            whill p <> END(M) do
                if RETRIEVE(p, M) .domain = d then
                    DELETE (p, M)
                    {удаляется элемент со значением d в поле domain}
                else
                    p:= NEXT(p, M) ;
            INSERT(x, FIRST(M), M)
                { пара (d, r) помещена в начало списка }
        end; { ASSIGN }

    function COMPUTE ( var M: MAPPING;
                d: domaintype; var r: rangetype ): boolean;
        var
            p: position;
```

```
begin
    p:= FIRST(M) ;
    whill p <> END (M) do begin
        if RETRIEVE(p, M) .domain = d then begin
            r:= RETRIEVE (p, M) .range;
            return(true)
        end;
        p:= NEXT(p, M)
    end;
    return(false) { d не принадлежит области определения }
end; { COMPUTE }
```

2.6. Стеки и рекурсивные процедуры

Стеки находят важное применение при реализации рекурсивных процедур в языках программирования. *Организация выполнения процедур* в языках программирования состоит в задании структур данных, которые используются для хранения значений программных переменных во время выполнения программы. Все языки программирования, допускающие рекурсивные процедуры, используют стеки *активационных записей* для хранения всех значений переменных, принадлежащих каждой активной процедуре. При вызове процедуры *P* новая активационная запись для этой процедуры помещается в стек независимо от того, есть ли в стеке другие активационные записи для процедуры *P*. Таким образом, извлекая активационную запись из стека для последнего вызова процедуры *P*, можно управлять возвратом к точке в программе, из которой *P* вызывалась (эта точка, называемая *адресом возврата,* помещается в активационную запись процедуры *P* при вызове этой процедуры).

Рекурсивные вызовы процедур упрощают структуру многих программ. Но в некоторых языках программирования процедурные вызовы более "дорогие" (по времени выполнения), чем непосредственное выполнение операторов, поэтому программа может работать быстрее, если из нее исключить рекурсивные процедуры. Здесь мы не выступаем в поддержку обязательного исключения рекурсивных или каких-либо иных процедур, очень часто структурная простота программы важна не менее, чем ее малое время выполнения. На практике бывают ситуации, когда после реализации части программного проекта возникает необходимость исключить рекурсию, и основная цель этого обсуждения заключается только в том, чтобы подвести к вопросу о том, как можно преобразовать рекурсивную процедуру в нерекурсивную с помощью стеков.

Пример 2.3. Рассмотрим рекурсивное и нерекурсивное решения упрощенной версии классической *задачи о ранце,* где мы имеем *целевое значение t* и конечное множество положительных целых *весов* $w_1, w_2, \dots, w_n$. Необходимо определить, можно ли из множества весов выбрать такой их набор, чтобы их сумма точно равнялась величине *t*. Например, если $t = 10$ и множество весов составляют числа 7, 5, 4, 4 и 1, то можно выбрать второй, третий и пятый веса, поскольку $5 + 4 + 1 = 10$.

В листинге 2.14 представлена функция *knapsack* (ранец), работающая с массивом весов *weights:* **array**[1..*n*] of integer. Функция *knapsack*(*s*, *i*) определяет, существует ли набор весов из множества весов от *weights*[*i*]до *weights*[*n*], сумма которых равна s, и если есть, то печатает этот набор весов. В частном случае, когда $s = 0$, пустое множество весов считается решением. Если $s < 0$ или $i > n$ (выход за заданное множество весов), то считаем, что решение не существует или не найдено (функция *knapsack* возвращает значение false).

Если ни один из перечисленных случаев к текущей ситуации не применим, то снова вызывается *knapsack*($s - w_i$, $i + 1$), чтобы посмотреть, существует ли решение, которое включает вес w_i. Если в этом случае решение существует, то оно является и решением исходной задачи; это решение, включая w_i, распечатывается. Если решения нет, то вызывается функция *knapsack*(s, $i + 1$) для поиска решения без участия веса w_i. П

Листинг 2.14. Рекурсивное решение задачи о ранце

```
     function knapsack ( target: integer; candidate: integer) :boolean;
         begin
(1)          if target:= 0 then
(2)              return(true)
(3)          else if (target < 0) or (candidate > л) then
(4)              return(false)
                 else { ищется решение с и без candidate }
(5)            if knapsack(target-weights[candidate]candidate+1)then
                     begin
(6)                      writeln(weights[candidate]);
(7)                      return(true)
                     end
                 else { возможное решение без candidate }
(8)              return(knapsack(target, candidate + 1))
         end; { knapsack }
```

Исключение „концевых" рекурсий

Часто можно чисто механически исключить последний вызов процедуры самой себя. Если процедура *P(x)* на последнем шаге вызывает *P(y),* то этот вызов можно заменить оператором присваивания *x* := *y* и последующим переходом в начало кода процедуры *P*. Здесь *y* может быть выражением, а *x* должно принимать значение, т.е. это значение хранится в локальной переменной. Если процедура имеет несколько параметров, то с ними надо поступить точно так же, как описано для *x и y*.

Описанная схема вызывает повторное выполнение процедуры *P* с новым значением параметра *x* с точно таким же конечным эффектом, какой был бы при рекурсивном вызове *P(y)* и возврате из этого вызова. Отметим, что тот факт, что некоторые локальные переменные процедуры *P* принимают новые значения, не имеет последствий, поскольку процедура уже не **должна** использовать старых значений этих переменных.

Подобный вариант исключения рекурсивного вызова можно проиллюстрировать на примере листинга 2.14, где последний шаг функции *knapsack* возвращает результат рекурсивного вызова без параметров. В такой ситуации также можно заменить вызов операторами присваивания значений параметрам и переходом в начало кода функции. В данном случае строку (8) в листинге 2.14 можно заменить следующими операторами:

```
candidate:= candidate + 1;
goto beginning
```

где *beginning* (начало) — метка на оператор строки (1). Отметим, что параметр *target* не переприсваивается, так как сохраняет старое значение. Поэтому проверки его значения в строках (1) и (3) фактически не нужны, необходимо только проверить условие, что *candidate* > *n* в строке (3).

Полное исключение рекурсий

Описанная выше техника исключения рекурсивных вызовов полностью удаляет рекурсии только тогда, когда рекурсивные вызовы находятся в конце процедур и вызов осуществляется в форме, позволяющей его исключить. Существует более общий подход, позволяющий преобразовать любую рекурсивную процедуру (или функцию) в нерекурсивную, но этот подход вводит определяемые пользователем стеки. В общем случае этот стек хранит следующее.

1. Текущие значения параметров процедуры.

2. Текущие значения всех локальных переменных процедуры.
3. Адрес возврата, т.е. адрес места, куда должно перейти управление после завершения процедуры.

В случае функции *knapsack* положение упрощается. Во-первых, заметим, что при всех вызовах (при этом происходит вставка записи в стек) параметр *candidate* увеличивается на единицу. Поэтому мы можем хранить значение *candidate* как глобальную переменную, значение которой увеличивается на единицу при вставке записи в стек и уменьшается на единицу при извлечении записи из стека.

Во-вторых, следующее упрощение можно сделать, если в стеке хранить модифицированный адрес возврата. Отметим, что адрес возврата для этой функции может находиться или в другой процедуре, вызвавшей *knapsack,* или в строке (5), или в строке (8). Можно представить эти три возможности с помощью "статуса", который может принимать три значения.

1. *none* (нет). Показывает, что вызов осуществлен из внешней процедуры.
2. *included* (включенный). Указывает на вызов из строки (5), которая включает вес ***weights*[*candidate*]** в возможное решение.
3. *excluded* (исключенный). Указывает на вызов из строки (8), которая исключает вес ***weights*[*candidate*]** из возможного решения.

Если мы сохраним статус как адрес возврата, то сможем рассматривать *target* как глобальную переменную. При изменении статуса с *none* на *included* мы вычитаем вес ***weights*[*candidate*]** из *target* и прибавляем его обратно при изменении статуса с *included* на *excluded.* Чтобы показать, что рекурсивно вызываемой *knapsack* найдено решение, используем глобальную переменную ***winflag*** (флаг победы). Получив один раз значение true, *winflag* сохраняет это значение, из стека извлекаются записи, и те веса, которые имеют статус *included,* распечатываются. В описанной ситуации стек можно объявить как список статусов (statuses) следующим способом:

```
type
    statuses = (none, included, excluded);
    STACK = { подходящее объявление стека }
```

В листинге 2.15 приведена нерекурсивная процедура *knapsack,* оперирующая с массивом весов *weights.* Хотя эта процедура может выполняться быстрее, чем исходная рекурсивная функция *knapsack,* но видно, что код ее длиннее и она труднее для понимания. Поэтому исключение рекурсий следует применять только тогда, когда скорость выполнения программы является решающим фактором.

Листинг 2.15. Нерекурсивная процедура *knapsack*

```
procedure knapsack ( target: integer );
    var
        candidate: integer;
        winflag: boolean;
        S: STACK;
    begin
        candidate:= 1;
        winflag:= false;
        MAKENULL(S);
        PUSH(none, S);
            { инициализация стека для рассмотрения weights[1] }
        repeat
            if winflag then begin
                    { извлечение записи из стека и
                            печать весов, входящих в решение }
                if TOP(S) = included then
```

```
                writeln(weights[candidate]);
            candidate:= candidate - 1;
            POP(S)
        end
        else if target = 0 then begin { решение найдено }
            winflag:= true;
            candidate:= candidate - 1;
            POP(S)
        end
        else if (((target < 0) and (TOP(S) = none))
                or (candidate > n)) then begin { решения нет }
            candidate:= candidate - 1;
            POP (S)
        end
        else { решения пока нет,
                рассматривается статус текущего кандидата }
        if TOP(S) = none then begin
                    { первая попытка включить кандидата }
            target:= target - weights[candidate];
            candidate:= candidate + 1;
            POP(S); PUSH(included, S); PUSH(none, S)
        end
        else if TOP(S) = included then begin
                    { попытка исключить кандидата }
            target:= target + weights[candidate];
            candidate:= candidate + 1;
            POP(S); PUSH(excluded, S); PUSH(none, S)
        end
        else begin { TOP(S) = excluded;
                        отказ от текущего выбора }
            POP(S);
            candidate:= candidate - 1
        end
    until EMPTY(S)
end; { knapsack }
```

Упражнения

2.1. **Напишите с помощью операторов списка программу печати элементов списка.**

2.2. **Напишите программы вставки, удаления и поиска элементов отсортированного списка, используя для реализации списка**

а) массив;

б) указатели;

в) курсоры.

Каково время выполнения каждой из этих программ?

2.3. **Напишите программу для слияния**

а) двух отсортированных списков;

б) и отсортированных списков.

2.4. **Напишите программу объединения списков в один список.**

2.5. **Рассмотрим многочлены вида $p(x) = c_1x^{e_1} + c_2x^{e_2} + \cdots + c_nx^{e_n}$, где $e_1 > e_2 > \ldots > e_n \geq 0$. Такой многочлен можно представить в виде связанного списка, где каждая ячейка имеет три поля: одно — для коэффициента c_i, второе — для показателя**

степени e_i, третье — для указателя на следующую ячейку. Для описанного представления многочленов напишите программу их дифференцирования.

2.6. Напишите программу сложения и умножения многочленов, используя их представление, описанное в упражнении 2.5.

*2.7. Пусть ячейки объявлены следующим образом:

```
type
    celltype = record
        bit: 0..1;
        next: ↑ celltype
    end;
```

Двоичное число $b_1 b_2 ... b_n$, где каждое b_i = 0, 1 имеет десятичное значение $\sum_{i=1}^{n} b_i 2^{n-1}$. Это число можно представить в виде списка $b_1, b_2, ..., b_n$. Данный список, в свою очередь, представим как связанный список ячеек типа celltype, определенного выше. Напишите процедуру ***increment(bnumber)***, которая прибавляет 1 к двоичному числу ***bnumber***. *Совет:* сделайте процедуру *increment* рекурсивной.

2.8. Напишите процедуру обмена элементами в позициях *p* и **NEXT(*p*)** для простого связанного списка.

*2.9. Следующая процедура предназначена для удаления всех вхождений элемента *x* в списке *L*. Найдите причину, по которой эта процедура будет выполняться не всегда, и устраните ее.

```
procedure delete ( x: elementtype; var L: LIST );
    var
        p: position;
    begin
        p:= FIRST(L);
        while p <> END(L) do begin
            if RETRIEVE(p, L) = x then
                DELETE(p, L);
            p:= NEXT(p, L)
        end
    end; { delete }
```

2.10. Необходимо сохранить список в массиве *A,* чьи ячейки содержат два поля: *data* — для элементов и поле *position* — для позиций (целых чисел) элементов. Целочисленная переменная *last* (последний) используется для указания того, что список содержится в ячейках от ***A*[1]** до ***A*[*last*]** массива *A*. Тип LIST определен следующим образом:

```
type
    LIST = record
        last: integer;
        elements: array[1..maxlength] of record
            data: elementtype;
            position: integer;
        end
    end;
```

Напишите процедуру **DELETE(*p*, *L*)** для удаления элемента в позиции *p*. Включите в процедуру все необходимые проверки на "внештатные" ситуации.

2.11. Пусть *L* — это список типа LIST, *p*, *q* и *r* — позиции. Определите как функцию от *n* (длины списка *L*) количество выполнений функций FIRST, END и NEXT в следующей программе

```
p: = FIRST(L);
while p <> END(L) do begin
    q:= p;
    while q <> END(L) do begin
        q:= NEXT(q, L);
        r:= FIRST(L);
        while r <> q do
            r:= NEXT(r, L)
    end;
    p:= NEXT(p, L)
end;
```

2.12. Перепишите код операторов, выполняемых над списками LIST, в предположении, что реализуются однонаправленные списки.

2.13. Добавьте необходимые проверки на ошибки в процедуры листинга 2.6.

2.14. Еще одна реализация списков посредством массивов совпадает с описанной в разделе 2.2 за исключением того, что при удалении элемента он заменяется значением "удален", которое другим способом в списке появиться не может. Перепишите операторы списка для этой реализации. Какие достоинства и недостатки этой реализации списков по сравнению с исходной?

2.15. Предположим, что мы хотим использовать логическую переменную при реализации очереди для указания того, что очередь пуста. Измените объявления и операторы в реализации очередей Посредством циклических массивов для использования этой переменной. Что можно ожидать от такой рационализации?

2.16. *Очередь с двусторонним доступом* — это список, в котором добавлять и удалять элементы можно с обоих концов. Разработайте реализации для таких очередей с использованием массивов, указателей и курсоров.

2.17. Определите АТД, поддерживающие операторы ENQUEUE, DEQUEUE (см. раздел 2.4) и ONQUEUE. **ONQUEUE(x)** — функция, возвращающая значения true или false в зависимости от того, присутствует или нет элемент x в очереди.

2.18. Как реализовать очередь, если элементами являются символьные строки произвольной длины? Сколько времени необходимо для операции вставки такого элемента в очередь?

2.19. В одной возможной реализации очередей посредством связанных списков не используется ячейка заголовка, а указатель *front* указывает непосредственно на первую ячейку. Если очередь пуста, тогда *front* = *rear* = nil. Напишите необходимые операторы для этой реализации очередей. Сравните эту реализацию с реализацией, описанной в разделе 2.4, по критериям времени выполнения, требуемого объема памяти и лаконичности (краткости и понятности) кода.

2.20. В одном варианте реализации очередей посредством циклических массивов записываются позиция первого элемента и длина очереди.

а) необходимо ли в этой реализации ограничивать длину очереди числом ***maxlength*** - 1?

б) напишите пять операторов, выполняемых над **очередями,** для этой реализации;

в) сравните эту реализацию с реализацией посредством циклических массивов, описанной в разделе 2.4.

2.21. Возможно хранение двух стеков в одном массиве, если один располагается в начале массива и растет к концу массива, а второй располагается в конце массива и растет к началу. Напишите процедуру **PUSH(x,** S) вставки элемента x в стек S, где S — один или другой стек из этих двух стеков. Включите все необходимые проверки в эту процедуру.

2.22. Можно хранить k стеков в одном массиве, если используется структура данных, показанная на рис. 2.12 для случая $k = 3$. В этом случае можно организовать вставку и удаление элементов для каждого стека так же, как описано в разделе 2.3. Но если случится, что при вставке элемента в стек i вершина TOP(i) совпадет с "дном" предыдущего стека **BOTTOM(i–1)**, то возникает необходимость переместить все стеки так, чтобы между каждой парой смежных стеков был зазор из пустых ячеек массива. В этом случае мы можем сделать все зазоры одинаковыми или пропорциональными длине соседнего с зазором стека (из теории следует: чем больше стек, тем вероятнее в ближайшем будущем его рост, а мы, естественно, хотим отсрочить следующую реорганизацию массива).

а) в предположении, что есть процедура ***reorganize*** (реорганизация), вызываемая при возникновении "конфликтов" между стеками, напишите код для пяти операторов стека;

б) напишите процедуру ***reorganize*** в предположении, что уже существует процедура ***makenewtops*** (сделать новые вершины), которая вычисляет ***newtop*[*i*]**, — новую позицию вершины стека i для всех i, $1 < i < k$. *Совет:* отметим, что стек i может перемещаться как вверх, так и вниз по **массиву.** Если новая позиция стека j захватывает старую позицию стека i, то стек i надо переместить раньше стека j. Рассматривая стеки в порядке 1, 2, ..., k, создадим еще стек "целей", каждая "цель" будет перемещать отдельный конкретный стек. **Если** стек i можно безопасно переместить, то перемещение **выполняется** и повторно рассматривается стек, чей номер находится в вершине стека целей. Если стек i нельзя переместить, не затрагивая других стеков, то его номер помещается в стек целей;

в) какая подходящая реализация для стека целей? Необходимо ли для этого использовать список из целых чисел или можно воспользоваться более простой реализацией?

г) реализуйте процедуру *makenewtops* так, чтобы пространство перед каждым стеком было пропорционально его текущей длине;

д) какие изменения надо сделать в схеме рис. **2.12,** чтобы ее можно было применить для очередей? А для произвольных списков?

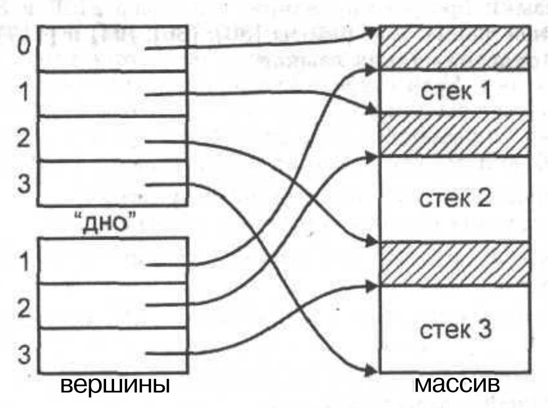


Рис. 2.12. Расположение нескольких стеков в одном массиве

2.23. Измените **реализации** операторов POP и ENQUEUE из разделов 2.3 и 2.4 так, чтобы они возвращали элементы, удаленные соответственно из стека и очере-

ди. Что необходимо сделать, если элемент имеет тип данных, который функция не может вернуть?

2.24. Используйте стек для исключения рекурсивных вызовов из следующих процедур:

•а)

```
function comb ( n, m: integer ): integer;
{вычисление биномиальных коэффициентов C™ при 0≤m≤n и n≥1}
    begin
        if (n = 1) or (m = 0) or (m = n) then
            return(1)
        else
            return(comb(n - 1, m) + comb(n - 1, m - 1))
    end; { comb }
```

б)

```
procedure reverse ( var L: LIST );
    { обращение списка L }
    var
        x: elementtype;
    begin
        if not EMPTY(L) then begin
            x:=RETRIEVE(FIRST(L),L);
            DELETE(FIRST(L), L);
            reverse(L);
            INSERT(x, END(L), L)
        end
    end; { reverse }
```

***2.25.** Можно ли исключить последние рекурсивные вызовы из программ примера 2.24? Если можно, то как?

Библиографические примечания

Книга [63] содержит дополнительный материал по реализациям списков, стеков и очередей. Многие языки программирования, например LISP и SNOBOL, поддерживают списки в удобной форме. См. работы [80], [86], [94] и [117], содержащие исторические справки и описания таких языков.

ГЛАВА 3

Деревья

Деревья представляют собой иерархическую структуру некой совокупности элементов. Знакомыми вам примерами деревьев могут служить генеалогические и организационные диаграммы. Деревья используются при анализе электрических цепей, при представлении структур математических формул. Они также естественным путем возникают во многих областях компьютерных наук. Например, деревья используются для организации информации в системах управления базами данных и для представления синтаксических структур в компиляторах программ. В главе 5 описано применение деревьев для представления данных. В этой книге вы встретитесь со многими примерами деревьев. В этой главе мы введем основные определения (терминологию) и представим основные операторы, выполняемые над деревьями. Затем опишем некоторые наиболее часто используемые структуры данных, представляющих деревья, и покажем, как наиболее эффективно реализовать операторы деревьев.

3.1. Основная терминология

Дерево — это совокупность элементов, называемых *узлами* (один из которых определен как *корень),* и отношений ("родительских"), образующих **иерархическую** структуру узлов. Узлы, так **же,** как и элементы списков, могут быть элементами любого типа. Мы часто будем изображать узлы буквами, строками или числами. Формально *дерево* можно рекуррентно определить следующим образом.

1. Один узел является деревом. Этот же узел также является корнем этого дерева.
2. Пусть п — это узел, а T_1, T_2 . . . .T_k — деревья с корнями n_1, n_2, ..., n_k соответственно. Можно построить новое дерево, сделав га родителем узлов n_1, га$_г$, ..., n_k. В этом дереве *n* будет корнем, а T_1, T_2, ..., T_k — *поддеревьями* этого корня. Узлы $n_1, n_2,$ • • • >n_kназываются ***сыновьями***узла га.

Часто в это определение включают понятие *нулевого дерева,* т.е. "дерева" без узлов, такое дерево мы будем обозначать символом Л.

Пример 3.1. Рассмотрим оглавление книги, схематически представленное на рис. 3.1, а. Это оглавление является деревом, которое в другой форме показано на рис. 3.1, б. Отношение родитель-сын отображается в виде линии. Деревья обычно рисуются сверху вниз, как на рис. **3.1,** б, так, что родители располагаются выше "детей".

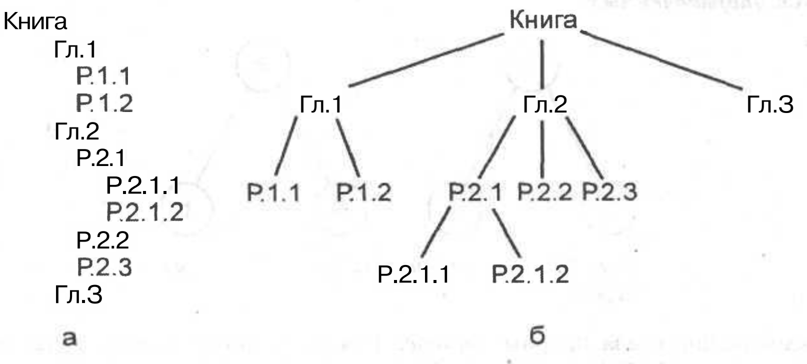


Рис. 3.1. Оглавление книги и его представление в виде дерева

Корнем этого дерева является узел *Книга,* который имеет три поддерева соответственно с корнями *Гл.1*, *Гл.2* и *Гл.3.* Эти отношения показаны линиями, идущими из корня *Книга* к узлам *Гл.1*, *Гл.2* и Гл.3. Узел *Книга* является родителем узлов *Гл.1*, *Гл.2* и Гл.3, а эти три узла — сыновьями узла *Книга.*

Третье поддерево, с корнем Гл.3, состоит из одного узла, остальные два поддерева имеют нетривиальную структуру. Например, поддерево с корнем *Гл.2 в* свою очередь имеет три поддерева, которые соответствуют разделам книги *Р.2.1*, *Р.2.2* и *Р.2.3*; последние два поддерева имеют по одному узлу, в то время как первое имеет два поддерева, соответствующие подразделам книги *Р.2.1.1* и *Р.2.1.2*. □

Пример **3.1** показывает типичные данные, для которых наилучшим представлением будут деревья. В этом примере родительские отношения устанавливают подчиненность глав и разделов: родительский узел связан с узлами сыновей (и указывает на них), как узел *Книга* подчиняет себе узлы *Гл.1*, *Гл.2* и Гл.3. В этой книге вы встретите много различных отношений, которые можно представить с помощью родительских отношений и подчиненности в виде деревьев.

Путем из узла n_1 *в узел* n_k называется последовательность узлов $n_1, n_2, \ldots, n_k$, где для всех i, $1 < i < k$, узел n_i является родителем узла n_{i+1}. *Длиной пути* называется число, на единицу меньшее числа узлов, составляющих этот путь. Таким образом, путем нулевой длины будет путь из любого узла к самому себе. На рис. 3.1 путем длины 2 будет, например, путь от узла *Гл.2* к узлу *Р.2.1.2.*

Если существует путь из узла о в ft, то в этом случае узел *а* называется *предком* узла *Ь,* а узел *Ь* — *потомком* узла *а.* Например, на рис. 3.1 предками узла *Р.2.1* будут следующие узлы: сам узел *Р.2.1* и узлы *Гл.2* и *Книга,* тогда как потомками этого узла являются опять сам узел *Р.2.1* и узлы *Р.2.1.1* и *Р.2.1.2.* Отметим, что любой узел одновременно является и предком, и потомком самого себя.

Предок или потомок узла, не являющийся таковым самого себя, называется *истинным предком* или *истинным потомком* соответственно. В дереве только корень не имеет **истинного** предка. Узел, не имеющий истинных потомков, называется *листом.* Теперь поддерево какого-либо дерева можно определить как узел (корень поддерева) вместе со всеми его потомками.

Высотой узла дерева называется длина самого длинного пути из этого узла до какого-либо листа. На **рис.3.1** высота узла *Гл.1* равна 1, узла *Гл.2* — 2, а узла Гл.3 — 0. *Высота дерева* совпадает с высотой корня. *Глубина узла* определяется как длина пути (он единственный) от корня до этого узла.

Порядок узлов

Сыновья узла обычно упорядочиваются слева направо. Поэтому два дерева на рис. 3.2 различны, так как порядок сыновей узла *а* различен. Если порядок сыновей игнорируется, то такое дерево называется *неупорядоченным,* в противном случае дерево называется *упорядоченным.*

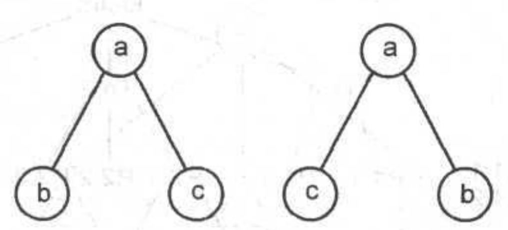


Рис. 3.2. Два различных упорядоченных дерева

Упорядочивание слева направо *сыновей* ("родных детей" одного узла) можно использовать для сопоставления узлов, которые не связаны отношениями предки-потомки. Соответствующее правило звучит следующим образом: если узлы *a* и *b* яв-

ляются сыновьями одного родителя и узел *a* лежит слева от узла *Ь,* то все потомки узла а будут находиться слева от любых потомков узла *Ъ*.

Пример 3.2. Рассмотрим дерево на рис. 3.3. Узел 8 расположен справа от узла 2, слева от узлов 9, 6, 10, 4 и 7, и не имеет отношений справа-слева относительно предков 1, 3 и 5.

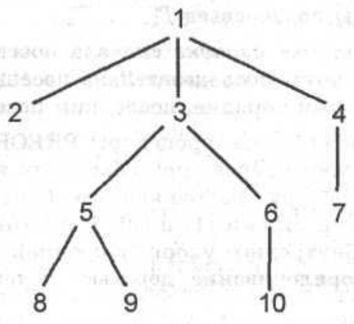


Рис. 3.3. *Дерево*

Существует простое правило, позволяющее определить, какие узлы расположены слева от данного узла га, а какие — справа. Для этого надо прочертить путь от корня дерева до узла га. Тогда все узлы и их потомки, расположенные слева от этого пути, будут находиться слева от узла *n,* и, аналогично, все узлы и их потомки, расположенные справа от этого пути, будут находиться справа от узла *n.* D

Прямой, обратный и симметричный обходы дерева

Существует несколько способов обхода (прохождения) всех узлов **дерева**[1]. Три наиболее часто используемых способа обхода дерева называются *обход в прямом порядке, обход в обратном порядке* и *обход во внутреннем порядке* (последний вид обхода также часто называют *симметричным обходом,* мы будем использовать оба этих названия как синонимы). Все три способа обхода рекурсивно можно определить следующим образом.

- Если дерево *T* является нулевым деревом, то в список обхода заносится пустая запись.
- Если дерево *T* состоит из одного узла, то в список обхода записывается этот узел.
- Далее, пусть *T* — дерево с корнем *n* и поддеревьями T_1, T_2, ..., T_k, как показано на рис. 3.4. Тогда для различных способов обхода имеем следующее.

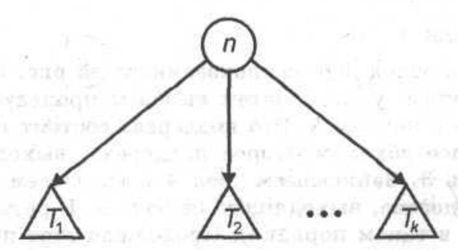


Рис. 3.4. Дерево T

[1] Обход узлов дерева равнозначен **упорядочиванию** по какому-либо правилу этих узлов. Поэтому в данном разделе мы будем использовать слова "обход узлов" и "упорядочивание узлов" как синонимы. — *Прим. ред.*

1. При *прохождении в прямом порядке* (т.е. при *прямом упорядочивании*) узлов дерева T сначала посещается корень n, затем узлы поддерева T_1, далее все узлы поддерева T_2, и т.д. Последними посещаются узлы поддерева T_k.
2. При *симметричном обходе* узлов дерева T сначала посещаются в симметричном порядке все узлы поддерева T_1, далее корень n, затем последовательно в симметричном порядке все узлы поддеревьев $T_2, \ldots, T_k$.
3. Во время *обхода в обратном порядке* сначала посещаются в обратном порядке все узлы поддерева T_1, затем последовательно посещаются все узлы поддеревьев $T_2, ..., T_k$, также в обратном порядке, последним посещается корень n.

В листинге 3.1,а показан набросок процедуры PREORDER (Прямое упорядочивание), составляющей список узлов дерева при обходе его в прямом порядке. Чтобы из этой процедуры сделать процедуру, выполняющую обход дерева в обратном порядке, надо просто поменять местами строки (1) и (2). В листинге 3.1,б представлен набросок процедуры INORDER (Внутреннее упорядочивание). В этих процедурах производится соответствующее упорядочивание деревьев путем вызова соответствующих процедур к корню дерева.

Листинг 3.1. Рекурсивные процедуры обхода деревьев

а. Процедура PREORDER

```
    procedure PREORDER ( n: узел );
        begin
(1)            занести в список обхода узел n;
(2)            for для каждого сына c узла n в порядке слева направо do
                   PREORDER(c)
        end; { PREORDER }
```

б. Процедура INORDER

```
    procedure INORDER ( n: узел );
        begin
            if n — лист then
                занести в список обхода узел n;
            else begin
                INORDER(самый левый сын узла n);
                занести в список обхода узел n;
                for для каждого сына c узла n, исключая самый левый,
                    в порядке слева направо do
                    INORDER(c)
            end
        end; { INORDER }
```

Пример 3.3. Сделаем обход дерева, показанного на рис. 3.3, в прямом порядке. Сначала запишем (посетим) узел 1, затем вызовем процедуру PREORDER для обхода первого поддерева с корнем 2. Это поддерево состоит из одного узла, которое мы и записываем. Далее обходим второе поддерево, выходящее из корня 1, это поддерево имеет корень 3. Записываем узел 3 и вызываем процедуру PREORDER для обхода первого поддерева, выходящего из узла 3. В результате получим список узлов 5, 8 и 9 (именно в таком порядке). Продолжая этот процесс, в конце мы получим полный список узлов, посещаемых при прохождении в прямом порядке исходного дерева: 1, 2, 3, 5, 8, 9, 6, 10, 4 и 7.

Подобным образом, предварительно преобразовав процедуру PREORDER в процедуру, выполняющую обход в обратном порядке (как указано выше), можно получить обратное упорядочивание узлов дерева из рис. 3.3 в следующем виде: 2, 8, 9, 5, 10,

6, 3, 7, 4 и 1. Применяя процедуру INORDER, получим список симметрично упорядоченных узлов этого же дерева: 2, 1, 8, 5, 9, 3, 10, 6, 7 и 4.

При обходе деревьев можно применить следующий полезный прием. Надо нарисовать непрерывный контур вокруг дерева, начиная от корня дерева, рисуя контур против часовой стрелки и поочередно обходя все наружные части дерева. Такой контур вокруг дерева из рис. 3.3 показан на рис. 3.5.

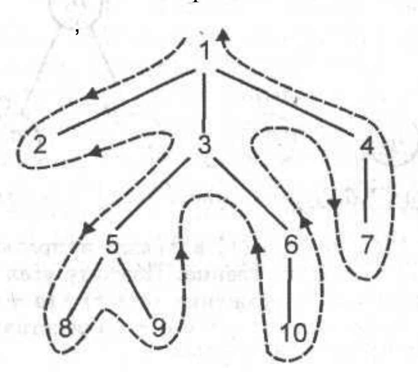


Рис. 3.5. *Контур дерева*

При прямом упорядочивании узлов надо просто записать их в соответствии с нарисованным контуром. При обратном упорядочивании после записи всех сыновей переходим к их родителю. При симметричном (внутреннем) упорядочивании после записи самого правого листа мы переходим не по ветви в направлении к корню дерева, а к следующему "внутреннему" узлу, который еще не записан. Например, если на рис. 3.5 узлы 2 и 1 уже записаны, то мы как бы перескакиваем "залив" между узлами 2 и 3 и переходим к узлу 8. Отметим, что при любом упорядочивании листья всегда записываются в порядке слева направо, при этом в случае симметричного упорядочивания между сыновьями может быть записан родитель. □

Помеченные деревья и деревья выражений

Часто бывает полезным сопоставить каждому узлу дерева *метку* (label) или значение, точно так же, как мы в предыдущей главе сопоставляли элементам списков определенные значения. Дерево, у которого узлам сопоставлены метки, называется *помеченным деревом.* Метка узла — это не имя узла, а значение, которое "хранится" в узле. В некоторых приложениях мы даже будем изменять значение метки, поскольку имя узла сохраняется постоянным. Полезна следующая аналогия: дерево—список, узел-позиция, метка-элемент.

Пример 3.4. На рис. 3.6 показано дерево с метками, представляющее арифметическое выражение *(а + Ь) * (а + с),* где n_1, ..., n_7 — имена узлов (метки на рисунке проставлены рядом с соответствующими узлами). Правила соответствия меток деревьев элементам выражений следующие.

1. Метка каждого листа соответствует операнду и содержит его значение, например узел n_4 представляет операнд *а.*
2. Метка каждого внутреннего (родительского) узла соответствует оператору. Предположим, что узел *n* помечен бинарным оператором в (например, + или *) и левый сын этого узла соответствует выражению E_1, а правый — выражению E_2. Тогда узел *n* и его сыновья представляют выражение $(E_1)\theta(E_2)$. Можно удалять родителей, если это необходимо.

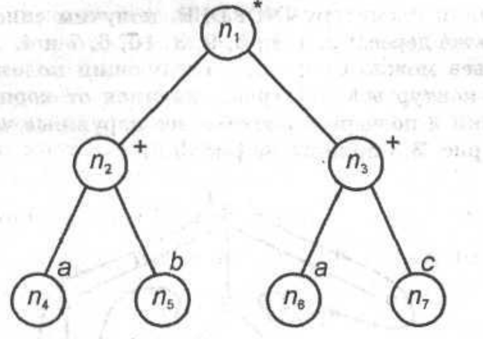


Рис. 3.6. Дерево выражения с метками

Например, узел n_2 имеет оператор +, а левый и правый сыновья представляют выражения (операнды) *а* и Ъ соответственно. Поэтому узел n_2 представляет (а) + (6), т.е. а + *Ь*. Узел n_1 представляет выражение (а + *Ь) * (а + с),* поскольку оператор * является меткой узла n_1, выражения $a + b$ и $a + c$ представляются узлами n_2 и n_3 соответственно. □

Часто при обходе деревьев составляется список не имен узлов, а их меток. В случае дерева выражений при прямом упорядочивании получаем известную *префиксную форму* выражений, где оператор предшествует и левому, и правому операндам. Для точного описания префиксной формы выражений сначала положим, что префиксным выражением одиночного операнда а является сам этот операнд. Далее, префиксная форма для выражения $(E_1)\theta\ (E_2)$, где θ — бинарный оператор, имеет вид θP_1P_2, здесь P_1 и P_2 — префиксные формы для выражений E_1 и E_2. Отметим, что в префиксных формах нет необходимости отделять или выделять отдельные префиксные выражения скобками, так как всегда можно просмотреть префиксное выражение θP_1P_2 и определить единственным образом P_1 как самый короткий префикс выражения P_1P_2.

Например, при прямом упорядочивании узлов (точнее, меток) дерева, показанного на рис. 3.6, получаем префиксное выражение **+ab+ac*. Самым коротким корректным префиксом для выражения *+ab+ac* будет префиксное выражение узла n_2: *+ab*.

Обратное упорядочивание меток дерева выражений дает так называемое *постфиксное* (или *польское) представление* выражений. Выражение θP_1P_2 в постфиксной форме имеет вид $P_1P_2\theta$, где P_1 и P_2 — постфиксные формы для выражений E_1 и E_2 соответственно. При использовании постфиксной формы также нет необходимости в применении скобок, поскольку для любого постфиксного выражения P_1P_2 легко проследить самый короткий суффикс P_2, что и будет корректным составляющим постфиксным выражением. Например, постфиксная форма выражения для дерева на рис. 3.5 имеет вид *ab+ac+**. Если записать это выражение как P_1P_2*, то P_2 (т.е. выражение *ас+)* будет самым коротким суффиксом для *ab+ac+* и, следовательно, корректным составляющим постфиксным выражением.

При симметричном обходе дерева выражений получим так называемую *инфиксную форму* выражения, которая совпадает с привычной "стандартной" формой записи выражений, но также не использует скобок. Для дерева на рис. 3.6 инфиксное выражение запишется как *а + Ь * а + с*. Читателю предлагается разработать алгоритм обхода дерева выражений, который бы выдавал инфиксную форму выражения со всеми необходимыми парами скобок.

Вычисление „наследственных" данных

Обход дерева в прямом или обратном порядке позволяет получить данные об отношениях предок-потомок узлов дерева. Пусть функция ***postorder(n)*** вычисляет позицию узла *n в* списке узлов, упорядоченных в обратном порядке. Например, для уз-

лов n_2, n_4 и га$_5$ дерева, представленного на рис. 3.6, значения этой функции будут 3, 1 и 2 соответственно. Определим также функцию *desc(n),* значение которой равно числу истинных потомков узла *n*.

Эти функции позволяют выполнить ряд полезных вычислений. Например, все узлы поддерева с корнем га будут последовательно занимать позиции от *postorder*(*n*) - *desc*(*n*) до *postorder*(*n*) в списке узлов, упорядоченных в обратном порядке. Для того чтобы узел *x* был потомком узла *y*, надо, чтобы выполнялись следующие неравенства:

$$postorder(y) - desc(y) < postorder(x) \leq postorderiy).$$

Подобные функции можно определить и для списков узлов, упорядоченных в прямом порядке.

3.2. Абстрактный тип данных TREE

В главе 2 списки, стеки, очереди и отображения получили трактовку как абстрактные типы данных (АТД). В этой главе рассмотрим деревья как **АТД** и как структуры данных. Одно из наиболее важных применений деревьев — это использование их при разработке реализаций различных АТД. Например, в главе 5 мы покажем, как можно использовать "дерево двоичного поиска" при реализации АТД, основанных на математической модели множеств. В следующих двух главах будут представлены многочисленные примеры применения деревьев при реализации различных абстрактных типов данных.

В этом разделе мы представим несколько полезных операторов, выполняемых над деревьями, и покажем, как использовать эти операторы в различных алгоритмах. Так же, как и в случае списков, можно предложить большой набор операторов, выполняемых над деревьями. Здесь мы рассмотрим следующие операторы.

1. **PARENT(*n*, *T*)**. Эта функция возвращает родителя (parent) узла *n* в дереве *T*. Если *n* является корнем, который не имеет родителя, то в этом случае возвращается Л. Здесь Л обозначает "нулевой узел" и указывает на то, что мы выходим за пределы дерева.
2. **LEFTMOST_CHILD(*n*, *T*)**. Данная функция возвращает самого левого сына узла *n* в дереве *T*. Если *n* является листом (и поэтому не имеет сына), то возвращается Л.
3. **RIGHT_SIBLING(*n*, *T*)**. Эта функция возвращает правого брата узла га в дереве *T* и значение Л, если такового не существует. Для этого находится родитель *p* узла га и все сыновья узла *p*, затем среди этих сыновей находится узел, расположенный непосредственно справа от узла *n*. Например, для дерева на рис. 3.6 **LEFTMOST_CHILD**(n_2) = n_4, **RIGHT_SIBLING**(n_4) = n_5 и **RIGHT_SIBLING**(n_5) = Л.
4. **LABEL(*n*, *T*)**. Возвращает метку узла га дерева *T*. Для выполнения этой функции требуется, чтобы на узлах дерева были определены метки.
5. **CREATE*i*(*v*, T_1, T_2, ..., T_i)** — это обширное семейство "созидающих" функций, которые для каждого i = О, 1, 2, ... создают новый корень *r* с меткой *v* и далее для этого корня создает i сыновей, которые становятся корнями поддеревьев T_1, T_2, ..., T_i. Эти функции возвращают дерево с корнем *r*. Отметим, что если i = О, то возвращается один узел *r*, который одновременно является и корнем, и листом.
6. **ROOT(*T*)** возвращает узел, являющимся корнем дерева *T*. Если *T* — пустое дерево, то возвращается Л.
7. **MAKENULL(*T*)**. Этот оператор делает дерево *T* пустым деревом.

Пример 3.5. Напишем рекурсивную PREORDER и нерекурсивную NPREORDER процедуры обхода дерева в прямом порядке и составления соответствующего списка его меток. Предположим, что для узлов определен тип данных node (узел), так же, как и для типа данных TREE (Дерево), причем АТД TREE определен для деревьев с

метками, которые имеют тип данных labeltype (тип метки). В листинге 3.2 приведена рекурсивная процедура, которая по заданному узлу *n* создает список в прямом порядке меток поддерева, корнем которого является узел п. Для составления списка всех узлов дерева *T* надо выполнить вызов **PREORDER(ROOT(*T*))**.

Листинг 3.2. Рекурсивная процедура обхода дерева в прямом порядке

```
procedure PREORDER ( n: node );
    var
        c: node;
    begin
        print(LABEL(n, T));
        c:= LEFTMOST_CHILD(n, T);
        while c o Л do begin
            PREORDER(c);
            C:= RIGHT_SIBLING(c, T)
        end
    end; { PREORDER }
```

Теперь напишем нерекурсивную процедуру для печати узлов дерева в прямом порядке. Чтобы совершить обход дерева, используем стек *S,* чей тип данных STACK уже объявлен как "стек для узлов". Основная идея разрабатываемого алгоритма заключается в том, что, когда мы дошли до узла и, стек хранит путь от корня до этого узла, причем корень находится на "дне" стека, а узел *n* — в вершине **стека.**[1]

Один из подходов к реализации обхода дерева в прямом порядке показан на примере программы NPREORDER в листинге 3.3. Эта программа выполняет два вида операций, т.е. может находиться как бы в одном из двух режимов. Операции первого вида (первый режим) осуществляют обход по направлению к потомкам самого левого еще не проверенного пути дерева до тех пор, пока не встретится лист, при этом выполняется печать узлов этого пути и занесение их в стек.

Во втором режиме выполнения программы осуществляется возврат по пройденному пути с поочередным извлечением узлов из стека до тех пор, пока не встретится узел, имеющий еще "не описанного" правого брата. Тогда программа опять переходит в первый режим и исследует новый путь, начиная с этого правого брата.

Программа начинается в первом режиме с нахождения корня дерева и определения, является ли стек пустым. В листинге 3.3 показан полный код этой программы. П

Листинг 3.3. Нерекурсивная процедура обхода дерева в прямом порядке

```
procedure NPREORDER ( T: TREE );
    var
        m: node; { переменная для временного хранения узлов }
        5: STACK; { стек узлов, хранящий путь от корня до
            родителя TOP(S) текущего узла т }
    begin { инициализация }
        MAKENULL(S);
        m:=ROOT(Г) ;
        while true do
            if m <> Л then begin
                print(LABEL(m, T));
                PUSH(m, S);
```

[1] Можно вернуться к разделу 2.6, где обсуждалась реализация рекурсивных процедур с помощью стека активационных записей. При рассмотрении листинга 3.2 нетрудно заметить, что активационную запись можно заносить в стек при каждом вызове процедуры **PREORDER(*n*)** и, это главное, стек будет содержать активационные записи для всех предков узла *n*.

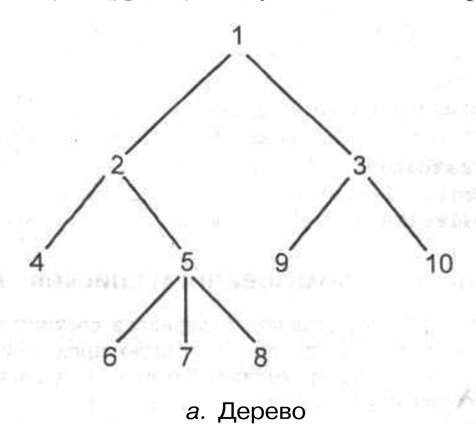
```
                { исследование самого левого сына узла т }
                m:= LEFTMOST_CHILD(m, T)
            end
            else begin
                {. завершена проверка пути, содержащегося в стеке }
                if EMPTY(S) then
                    return;
                { исследование правого брата узла,
                    находящегося в вершине стека }
                m:= RIGHT_SIBLING(TOP(S) , T);
                POP(S)
            end
    end; { NPREORDER }
```

3.3. Реализация деревьев

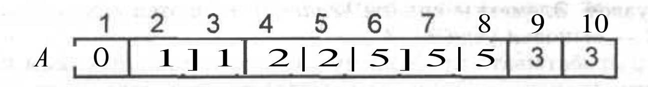
В этом разделе мы представим несколько основных реализаций деревьев и обсудим их возможности для поддержки операторов, введенных в разделе 3.2.

Представление деревьев с помощью массивов

Пусть T — дерево с узлами 1, 2, ..., n. Возможно, самым простым представлением дерева T, поддерживающим оператор PARENT (Родитель), будет линейный массив A, где каждый элемент $A[i]$ является указателем или курсором на родителя узла i. Корень дерева T отличается от других узлов тем, что имеет нулевой указатель или указатель на самого себя как на родителя. В языке Pascal указатели на элементы массива недопустимы, поэтому мы будем использовать схему с курсорами, тогда $A[i] = j$, если узел j является родителем узла i, и $A[i] = 0$, если узел i является корнем.



а. Дерево



	1	2	3	4	5	6	7	8	9	10
A	0	1	1	2	2	5	5	5	3	3

б. Курсоры на родителей

Рис. 3.7. Дерево и курсоры на родителей

Данное представление использует то свойство деревьев, что каждый узел, отличный от **корня,** имеет только одного родителя. Используя это представление, родителя любого узла можно найти за фиксированное время. Прохождение по любому пути, т.е. переход по узлам от родителя к родителю, можно выполнить за время, пропорциональное количеству узлов пути. Для реализации оператора LABEL можно использовать другой массив *L,* в котором элемент *L*[*i*] будет хранить метку узла *i,* либо объявить элементы массива *А* записями, состоящими из целых чисел (курсоров) и меток.

Пример 3.6. На рис. 3.7 показаны дерево и массив *А* курсоров на родителей этого дерева. D

Использование указателей или курсоров на родителей не помогает в реализации операторов, требующих информацию о сыновьях. Используя описанное представление, крайне тяжело для данного узла *n* найти его сыновей или определить его высоту. Кроме того, в этом случае невозможно определить порядок сыновей узла (т.е. какой сын находится правее или левее другого сына). Поэтому нельзя реализовать операторы, подобные LEFTMOST_CHILD и **RIGHT_SIBLING.** Можно ввести искусственный порядок нумерации узлов, например нумерацию сыновей в возрастающем порядке слева направо. Используя такую нумерацию, можно реализовать оператор **RIGHT_SIBLING,** код для этого оператора приведен в листинге **3.4.** Для задания типов данных node (узел) и TREE (Дерево) используется следующее объявление:

```
type
    node = integer;
    TREE = array [1..maxnodes] of node;
```

В этой реализации мы предполагаем, что нулевой узел Л представлен 0.

Листинг 3.4. Оператор определения правого брата

```
procedure RIGHT_SIBLING ( л: node; T: TREE ) : node;
    var
        i, parent: node;
    begin
        parent:= T[n];
        for i:= n + 1 to maxnodes do
            if T[i] = parent then
                return (i) ;
            return(0) { правый брат не найден }
    end; { RIGHT_SIBLING }
```

Представление деревьев с использованием списков сыновей

Важный и полезный способ представления деревьев состоит в формировании для каждого узла списка его сыновей. Эти списки можно представить любым методом, описанным в главе 2, но, так как число сыновей у разных узлов может быть разное, чаще всего для этих целей применяются связанные списки.

На рис. 3.8 показано, как таким способом представить дерево, изображенное на рис. 3.7, а. Здесь есть массив ячеек заголовков, индексированный номерами (они же имена) узлов. Каждый заголовок *(header)* указывает на связанный список, состоящий из **"элементов"-узлов.** Элементы списка *header*[*i*] являются сыновьями узла i, например узлы 9 и 10 — сыновья узла **3.**

Прежде чем разрабатывать необходимую структуру данных, нам надо в терминах абстрактного типа данных LIST (список узлов) сделать отдельную реализацию списков сыновей и посмотреть, как эти абстракции согласуются между собой. Позднее мы увидим, какие упрощения можно сделать в этих реализациях. Начнем со следующих объявлений типов:

```
type
    node = integer;
    LIST = { соответствующее определение для списка узлов };
    position = {соответствующее определение позиций в списках};
    TREE = record
        header: array[1..maxnodes] of LIST;
        labels: array[1..maxnodes] of labeltype;
        root: node
    end;
```

Рис. 3.8. Представление дерева с помощью связанных списков

Мы предполагаем, что корень каждого дерева хранится отдельно в поле *root* (корень). Для обозначения нулевого узла используется 0.

В листинге 3.5 представлен код функции LEFTMOST_CHILD. Читатель в качестве упражнения может написать коды других операторов.

Листинг 3.5. Функция нахождения самого левого сына

```
function LEFTMOST_CHILD ( n: node; T: TREE ): node;
    { возвращает самого левого сына узла л дерева T }
    var
        L: LIST; { скоропись для списка сыновей узла л }
    begin
        L:= T.header[n];
        if EMPTY(L) then { л является листом }
            return(0)
        else
            return(RETRIEVE(FIRST(L), L))
    end; { LEFTMOST_CHILD }
```

Теперь представим конкретную реализацию списков, где типы LIST и position имеют тип целых чисел, последние используются как курсоры в массиве записей *cellspace* (область ячеек):

```
var
    cellspace: array[1..maxnodes] of record
        node: integer;
        next: integer
    end;
```

Для упрощения реализации можно положить, что списки сыновей не имеют ячеек заголовков. Точнее, мы поместим *T.header*[*n*] непосредственно в первую ячейку списка, как показано на рис. 3.8. В листинге 3.6 представлен переписанный (с учетом этого упрощения) код функции LEFTMOST_CHILD, а также показан код функции PARENT, использующий данное представление списков. Эта функция более трудна для реализации, так как определение списка, в котором находится заданный узел, требует просмотра всех списков сыновей.

Листинг 3.6. функции, использующие представление деревьев посредством связанных списков

```
function LEFTMOST_CHILD ( n: node; T: TREE ): node;
    { возвращает самого левого сына узла n дерева T }
    var
        L: integer; { курсор на начало списка сыновей узла n }
    begin
        L:= T.header[n] ;
        if L = 0 then  { л  является листом }
            return(0)
        else
            return(cellspace[L].node)
    end; { LEFTMOST_CHILD }

function PARENT  ( n: node; T: TREE ): node;
    { возвращает родителя узла л дерева T }
    var
        p: node; { пробегает возможных родителей узла n }
        i: position; { пробегает список сыновей p }
    begin
        for p:= 1 to maxnodes do begin
            i:= T.header[p];
            while i <> 0 do { проверка на наличие сыновей узла p }
                if cellspace[i].node = л then
                    return(p)
                else
                    i:= cellspace[i].next
        end;
        return(0) { родитель не найден }
    end; { PARENT }
```

Представление левых сыновей и правых братьев

Среди прочих недостатков описанная выше структура данных не позволяет также с помощью операторов CREATE*i* создавать большие деревья из малых. Это является следствием того, что все деревья совместно используют массив *cellspace* для представления связанных списков сыновей; по сути, каждое дерево имеет собственный массив заголовков для своих узлов. А при реализации, например, оператора CREATE2(v, T_1, T_2) надо скопировать деревья T_1 и T_2 в третье дерево и добавить новый узел с меткой v и двух его сыновей — корни деревьев T_1 и T_2.

Если мы хотим построить большое дерево на основе нескольких малых, то желательно, чтобы все узлы всех деревьев располагались в одной общей области. Логическим продолжением представления дерева, показанного на рис. 3.8, будет замена массива заголовков на отдельный массив *nodespace* (область узлов), содержащий записи с произвольным местоположением в этом массиве. Содержимое поля *header* этих записей соответствует "номеру" узла, т.е. номеру записи в массиве *cellspace*, в

свою очередь поле ***node*** массива *cellspace* теперь является курсором для массива ***nodespace***, указывающим позицию узла. Тип TREE в этой ситуации просто курсор в массиве *nodespace,* указывающий позицию корня.

Пример 3.7. На рис. 3.9, а показано дерево, а на рис. 3.9, б — структура данных, где узлы этого дерева, помеченные как *A, B, C* и *D,* размещены в произвольных позициях массива *nodespace.* В массиве *cellspace* также в произвольном порядке размещены списки сыновей. □

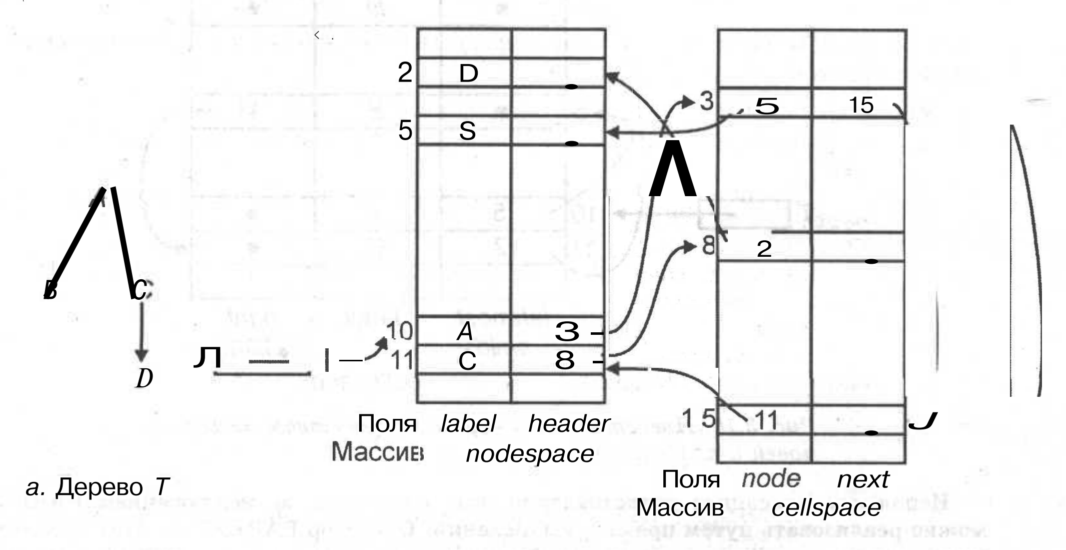


Рис. 3.9. Структура данных для дерева, использующая связанные списки

Структура данных, показанная на рис. 3.9, б, уже подходит для того, чтобы организовать слияние деревьев с помощью операторов CREATEi. Но и эту структуру можно значительно упростить. Для этого заметим, что цепочка указателей поля *next* массива *cellspace* перечисляет всех правых братьев.

Используя эти указатели, можно найти самого левого сына следующим образом. Предположим, что ***cellspace[i].node***— *n.* (Повторим, что "имя" узла, в отличие от его метки, является индексом в массиве *nodespace* и этот индекс записан в поле ***cellspace[i].node.***) Тогда указатель ***nodespace[n].header*** указывает на ячейку самого левого сына узла *n* в массиве *cellspace,* поскольку поле *node* этой ячейки является именем этого узла в массиве *nodespace.*

Можно упростить структуру, если идентифицировать узел не с помощью индекса в массиве *nodespace,* а с помощью индекса ячейки в массиве *cellspace,* который соответствует данному узлу как сыну. Тогда указатель *next* (переименуем это поле в ***right_sibling***— правый брат) массива *cellspace* будет точно указывать на правого брата, а информацию, содержащуюся в массиве *nodespace,* можно перенести в новое поле ***leftmost_child***(самый левый сын) массива *cellspace.* Здесь тип TREE является целочисленным типом и используется как курсор в массиве *cellspace,* указывающий на корень дерева. Массив *cellspace* можно описать как следующую структуру:

```
var
    cellspace: array[1..maxnodes] of record
        label: labeltype;
        leftmost_child: integer;
        right_sibling: integer
    end;
```

Пример 3.8. Новое представление для дерева, показанного на рис. 3.9, а, схематически изображено на рис. 3.10. Узлы дерева расположены в тех же ячейках массива, что и на рис. 3.9, б. □

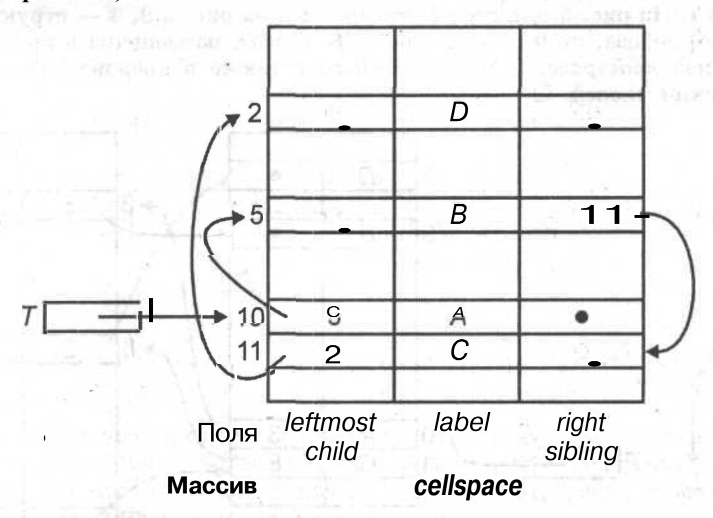


Рис. 3.10. Представление дерева посредством левых сыновей и правых братьев

Используя описанное представление, все операторы, за исключением PARENT, можно реализовать путем прямых вычислений. Оператор PARENT требует просмотра всего массива *cellspace*. Если необходимо эффективное выполнение оператора PARENT, то можно добавить четвертое поле в массив *cellspace* для непосредственного указания на родителей.

В качестве примера операторов, использующих структуры данных рис. 3.10, напишем код функции CREATE2, показанный в листинге 3.7. Здесь мы предполагаем, что неиспользуемые ячейки массива *cellspace* связаны в один свободный список, который в листинге назван как *avail*, и ячейки этого списка связаны посредством поля *right_sibling*. На рис. 3.11 показаны старые (сплошные линии) и новые (пунктирные линии) указатели в процессе создания нового дерева.

Листинг 3.7. функция CREATE2

```
function CREATE2 ( v. labeltype; T1, T2: integer ): integer;
    { возвращает новое дерево с корнем v и поддеревьями 71 и T2 }
    var
        temp: integer; { хранит индекс первой свободной ячейки
                                для корня нового дерева }
    begin
        temp:= avail;
        avail:= cellspace[avail].right_sibling;
        cellspace[temp].leftmost_child:= T1;
        cellspace[temp].label:= v;
        cellspace[temp].right_sibling:= 0;
        cellspace[T1] . right_sibling:= T2;
        cellspace[T2] . right_sibling:= 0; {необязательный оператор}
        return(temp)
    end; { CREATE2 }
```



Рис. 3.11. Изменение указателей при выполнении функции CREATE2

Можно уменьшить область памяти, занимаемую узлами дерева (но при этом увеличится время выполнения операторов), если в поле ***right_sibling*** самого правого сына вместо нулевого указателя поместить указатель на родителя. Но в этом случае, чтобы избежать двусмысленности, необходимо в каждую ячейку поместить еще двоичную (логическую) переменную, которая будет показывать, что содержится в поле ***right_sibling***: указатель на правого брата или указатель на родителя.

При такой реализации можно найти для заданного узла его родителя, следуя за указателями поля ***right_sibling***, пока не встретится указатель на родителя. В этом случае время, необходимое для поиска родителя, пропорционально количеству сыновей у родителя.

3.4. Двоичные деревья

Деревья, которые мы определили в разделе 3.1, называются *упорядоченными ориентированными деревьями,* поскольку сыновья любого узла упорядочены слева направо, а пути по дереву ориентированы от начального узла пути к его потомкам. *Двоичное* (или *бинарное) дерево* — совершенно другой вид. Двоичное дерево может быть или пустым деревом, или деревом, у которого любой узел или не имеет сыновей, или имеет либо *левого сына,* либо *правого сына,* либо обоих. Тот факт, что каждый сын любого узла определен как левый или как правый сын, существенно отличает двоичное дерево от упорядоченного ориентированного дерева.

Пример 3.9. Если мы примем соглашение, что на схемах двоичных деревьев левый сын всегда соединяется с родителем линией, направленной влево и вниз от родителя, а правый сын — линией, направленной вправо и вниз, тогда на рис. 3.12,а, б представлены два различных дерева, хотя они оба похожи на обычное (упорядоченное ориентированное) дерево, показанное на рис. 3.13. Пусть вас не смущает тот факт, что деревья на рис. 3.12,а, б различны и не эквивалентны дереву на рис. 3.13. Дело в том, что двоичные деревья нельзя непосредственно сопоставить обычному дереву. Например, на рис. 3.12,а узел 2 является левым сыном узла 1 и узел 1 не имеет правого сына, тогда как на рис. 3.12,б узел 1 не имеет левого сына, а имеет правого (узел 2). В тоже время в обоих двоичных деревьях узел 3 является левым сыном узла 2, а узел 4 — правым сыном того же узла 2. □

Обход двоичных деревьев в прямом и обратном порядке в точности соответствует таким же обходам обычных деревьев. При симметричном обходе двоичного дерева с

корнем n левым поддеревом T_1 и правым поддеревом T_2 сначала проходится поддерево T_1, затем корень n и далее поддерево T_2. Например, симметричный обход дерева на рис. 3.12,а даст последовательность узлов 3, 5, 2, 4, 1.

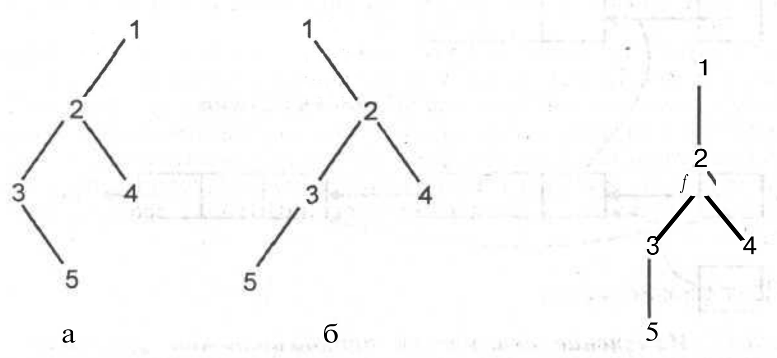


Рис. 3.12. Два двоичных дерева

Рис. 3.13. "Обычное" дерево

Представление двоичных деревьев

ЕСЛИ именами узлов двоичного дерева являются их номера 1, 2, ..., и, то подходящей структурой для представления этого дерева может служить массив ***cellspace*** записей с полями ***leftchild***(левый сын) и ***rightchild*** (правый сын), объявленный следующим образом:

```
var
    cellspace: array[1..maxnodes] of record
        leftchild: integer;
        rightchild: integer
    end;
```

В этом представлении ***cellspace***[*i*].***leftchild*** является левым сыном узла ***i***, а ***cellspace***[*i*].***rightchild*** — правым сыном. Значение 0 в обоих полях указывает на то, что узел *i* не имеет сыновей.

Пример 3.10. Двоичное дерево на рис. 3.12,а можно представить в виде табл. 3.1. П

Таблица 3.1. Представление двоичного дерева

	Значение поля *leftchild*	Значение поля *rightchild*
1	2	0
2	3	4
3	0	5
4	0	0
5	0	0

Пример: коды Хаффмана

Приведем пример применения двоичных деревьев в качестве структур данных. Для этого рассмотрим задачу конструирования *кодов Хаффмана.* Предположим, мы имеем сообщения, состоящие из последовательности символов. В каждом сообщении символы независимы и появляются с известной вероятностью, не зависящей от позиции в сообщении. Например, мы имеем сообщения, состоящие из пяти символов *a, Ь, c, d, e,* которые появляются в сообщениях с вероятностями 0.12, 0.4, 0.15, 0.08 и 0.25 соответственно.

Мы хотим закодировать каждый символ последовательностью из нулей и единиц так, чтобы код любого символа являлся префиксом кода сообщения, состоящего из последующих символов. Это *префиксное свойство* позволяет декодировать строку из нулей и единиц последовательным удалением префиксов (т.е. кодов символов) из этой строки.

Пример 3.11. В табл. 3.2 показаны две возможные кодировки для наших пяти символов. Ясно, что первый код обладает префиксным свойством, поскольку любая последовательность из трех битов будет префиксом для другой последовательности из трех битов; другими словами, любая префиксная последовательность однозначно идентифицируется символом. Алгоритм декодирования для этого кода очень прост: надо поочередно брать по три бита и преобразовать каждую группу битов в соответствующие символы. Например, последовательность 001010011 соответствует исходному сообщению *bcd*.

Таблица 3.2. Два двоичных кода

Символ	Вероятность	Код 1	Код 2
a	0.12	000	000
b	0.40	001	11
c	0.15	010	01
d	0.08	011	001
e	0.25	100	10

Легко проверить, что второй код также обладает префиксным свойством. Процесс декодирования здесь не отличается от аналогичного процесса для первого кода. Единственная сложность для второго кода заключается в том, что нельзя сразу всю последовательность битов разбить на отдельные сегменты, соответствующие символам, так как символы могут кодироваться и двумя и тремя битами. Для примера рассмотрим двоичную последовательность 1101001, которая опять представляет символы *bcd*. Первые два бита 11 однозначно соответствуют символу *b*, поэтому их можно удалить, тогда получится 01001. Здесь 01 также однозначно определяет символ *c* и т.д. □

Задача конструирования кодов Хаффмана заключается в следующем: имея множество символов и значения вероятностей их появления в сообщениях, построить такой код с префиксным свойством, чтобы средняя длина кода (в вероятностном смысле) последовательности символов была минимальной. Мы хотим минимизировать среднюю длину кода для того, чтобы уменьшить длину вероятного сообщения (т.е. чтобы сжать сообщение). Чем короче среднее значение длины кода символов, тем короче закодированное сообщение. В частности, первый код из примера 3.11 имеет среднюю длину кода 3. Это число получается в результате умножения длины кода каждого символа на вероятность появления этого символа. Второй код имеет среднюю длину 2.2, поскольку символы *a* и *d* имеют суммарную вероятность появления 0.20 и длина их кода составляет три бита, тогда как другие символы имеют код длиной 2[1].

Можно ли придумать код, который был бы лучше второго кода? Ответ положительный: существует код с префиксным свойством, средняя длина которого равна 2.15. Это наилучший возможный код с теми же вероятностями появления символов. Способ нахождения оптимального префиксного кода называется *алгоритмом Хаффмана*. В этом алгоритме находятся два символа a и b с наименьшими вероятностями появления и заменяются одним фиктивным символом, например *x*, который имеет вероятность появления, равную сумме вероятностей появления символов *a* и *b*. Затем, используя эту процедуру рекурсивно, находим оптимальный префиксный код для

[1] Отсюда следует очевидный вывод, что символы с большими вероятностями появления должны иметь самые короткие коды. — *Прим. ред.*

меньшего множества символов (где символы *a* и *b* заменены одним символом *x*). Код для исходного множества символов получается из кодов замещающих символов путем добавления 0 и 1 перед кодом замещающего символа, и эти два новых кода принимаются как коды заменяемых символов. Например, код символа *a* будет соответствовать коду символа *x* с добавленным нулем перед этим кодом, а для кода символа *b* перед кодом символа *x* будет добавлена единица.

Можно рассматривать префиксные коды как пути на двоичном дереве: прохождение от узла к его левому сыну соответствует 0 в коде, а к правому сыну — 1. Если мы пометим листья дерева кодируемыми символами, то получим представление префиксного кода в виде двоичного дерева. Префиксное свойство гарантирует, что нет символов, которые были бы метками внутренних узлов дерева (не листьев), и наоборот, помечая кодируемыми символами только листья дерева, мы обеспечиваем префиксное свойство кода этих символов.

Пример 3.12. Двоичные деревья для кодов 1 и 2 из табл. 3.2 показаны на рис. 3.14 (дерево слева соответствует коду 1, а дерево справа — коду 2). □

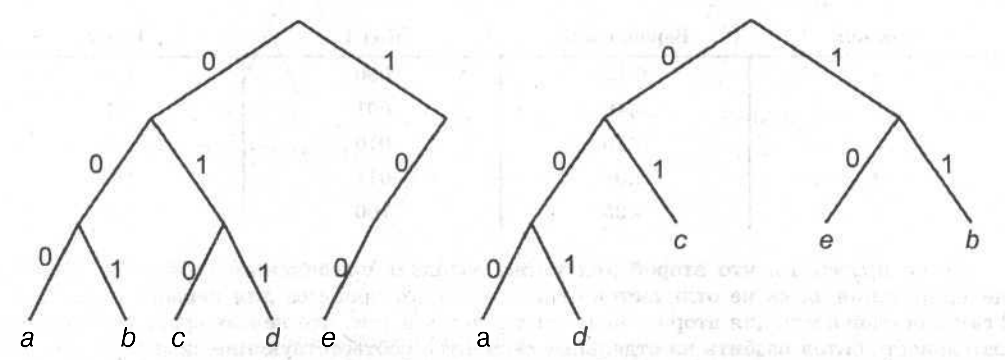
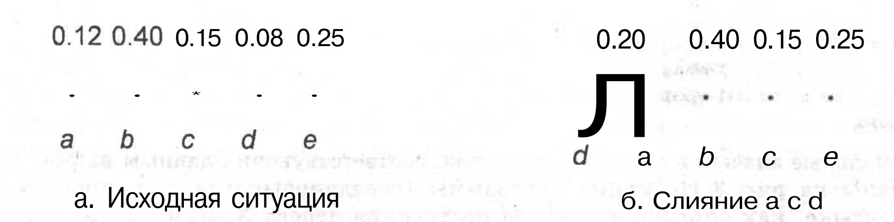


Рис. 3.14. Двоичные деревья, представляющие коды с префиксным свойством

Для реализации алгоритма Хаффмана мы используем *лес,* т.е. совокупность деревьев, чьи листья будут помечены символами, для которых разрабатывается кодировка, а корни помечены суммой вероятностей всех символов, соответствующих листьям дерева. Мы будем называть эти суммарные вероятности *весом* дерева. Вначале каждому символу соответствует дерево, состоящее из одного узла, в конце работы алгоритма мы получим одно дерево, все листья которого будут помечены кодируемыми символами. В этом дереве путь от корня к любому листу представляет код для символа-метки этого листа, составленный по схеме, согласно которой левый сын узла соответствует 0, а правый — 1 (как на рис. 3.14).

Важным этапом в работе алгоритма является выбор из леса двух деревьев с наименьшими весами. Эти два дерева комбинируются в одно с весом, равным сумме весов составляющих деревьев. При слиянии деревьев создается новый узел, который становится корнем объединенного дерева и который имеет в качестве левого и правого сыновей корни старых деревьев. Этот процесс продолжается до тех пор, пока не получится только одно дерево. Это дерево соответствует коду, который при заданных вероятностях имеет минимально возможную среднюю длину.

Пример 3.13. Последовательные шаги выполнения алгоритма Хаффмана для кодируемых символов и их вероятностей, заданных в табл. 3.2, представлены на рис. 3.15. Здесь видно (рис. 3.15,д), что символы *a, b, c, d* и *e* получили соответственно коды 1111, 0, 110, 1110 и 10. В этом примере существует только одно нетривиальное дерево, соответствующее оптимальному коду, но в общем случае их может быть несколько. Например, если бы символы *b* и *e* имели вероятности соответственно 0.33 и 0.32, то после шага алгоритма, показанного на рис. 3.15,в, можно было бы комбинировать *b* и *e,* а не присоединять *e* к большому дереву, как это сделано на рис. 3.15,г. □



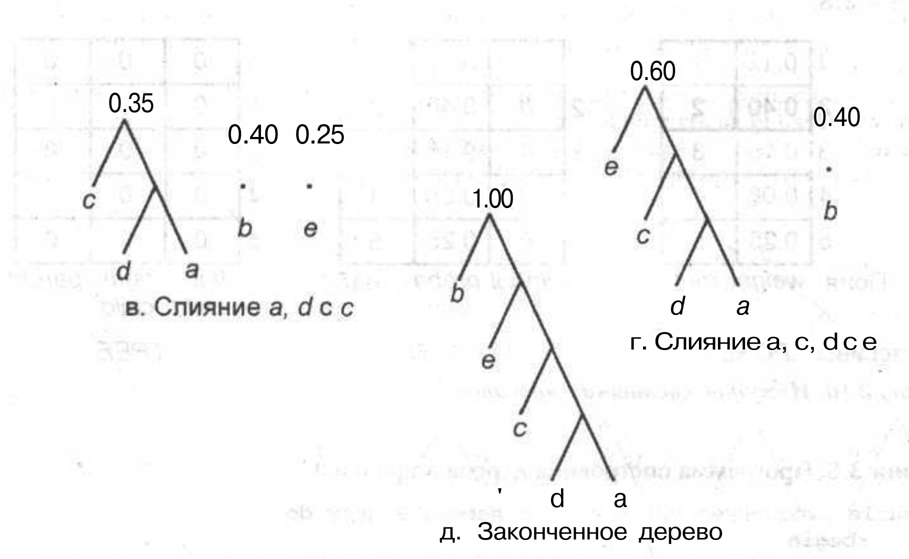


Рис. 3.15. Этапы создания дерева Хаффмана

Теперь опишем необходимые структуры данных. Во-первых, для представления двоичных деревьев мы будем использовать массив *TREE* (Дерево), состоящий из записей следующего типа:

```
record
    leftchild: integer;
    rightchild: integer;
    parent: integer
end
```

Указатели в поле *parent* (родитель) облегчают поиск путей от листа к корню при записи кода символов. Во-вторых, мы используем массив *ALPHABET* (Алфавит), также состоящий из записей, которые имеют следующий тип:

```
record
    symbol: char;
    probability: real;
    leaf: integer { курсор }
end
```

В этом массиве каждому символу (поле *symbol),* подлежащему кодированию, ставится в соответствие вероятность его появления (поле *probability)* и лист, меткой которого он является (поле *leaf).* В-третьих, для представления непосредственно деревьев необходим массив *FOREST* (Лес). Этот массив будет состоять из записей с полями *weight* (вес) и *root* (корень) следующего типа:

```
record
    weight: real;
    root: integer
end
```

Начальные значения всех трех массивов, соответствующих данным на рис. 3.15,а, показаны на рис. 3.16. Эскиз программы (псевдопрограмма, т.е. программа на псевдоязыке, как описано в главе 1) построения дерева Хаффмана представлен в листинге 3.8.

Массив *FOREST*

	weight	root
1	0.12	1
2	0.40	2
3	0.15	3
4	0.08	4
5	0.25	5

Массив *ALPHABET*

	symbol	probability	leaf
1	a	0.12	1
2	b	0.40	2
3	c	0.15	3
4	d	0.08	4
5	e	0.25	5

Массив *TREE*

	left-child	right-child	parent
1	0	0	0
2	0	0	0
3	0	0	0
4	0	0	0
5	0	0	0

Рис. 3.16. Исходное состояние массивов

Листинг 3.8. Программа построения дерева Хаффмана

```
(1) while существует более одного дерева в лесу do
        begin
(2)         i:= индекс дерева в FOREST с наименьшим весом;
(3)         j:= индекс дерева в FOREST со вторым наименьшим весом;
(4)         Создание нового узла с левым сыном FOREST[i].rootи
                правым сыном FOREST[j].root;
(5)         Замена в FOREST дерева i деревом, чьим корнем является
                новый узел и чей вес равен
                FOREST[i].weight + FOREST[j].weight;
(в)         Удаление дерева j из массива FOREST
        end;
```

Для реализации строки (4) листинга 3.8, где увеличивается количество используемых ячеек массива *TREE,* и строк (5) и (6), где уменьшается количество ячеек массива *FOREST,* мы будем использовать курсоры ***lasttree*** (последнее дерево) и ***lastnode*** (последний узел), указывающие соответственно на массив *FOREST* и массив *TREE*. Предполагается, что эти курсоры располагаются в первых ячейках соответствующих массивов[1]. Мы также предполагаем, что все массивы имеют определенную объявленную длину, но здесь мы не будем проводить сравнение этих ограничивающих значений со значениями курсоров.

В листинге 3.9 приведены коды двух полезных процедур. Первая из них, ***lightones***, выполняет реализацию строк (2) и (3) листинга 3.8 по выбору индексов двух деревьев с наименьшими весами. Вторая процедура, функция ***create***(n_1, n_2), создает новый узел и делает заданные узлы n_1 и n_2 левым и правым сыновьями этого узла.

---

[1] Для этапа чтения данных, который мы опускаем, необходим также курсор для массива *ALPHABET,* который бы "следил" за заполнением данными этого массива.

Листинг 3.9. Две процедуры

```
procedure lightones ( var least, second: integer );
    { присваивает переменным least и second индексы массива
        FOREST, соответствующие деревьям с наименьшими весами.
        Предполагается, что lasttree > 2. }
    var
        i: integer;
    begin { инициализация least и second,
                рассматриваются первые два дерева }
        if FOREST[1].weight <= FOREST[2].weight then
            begin least:= 1; second:= 2 end
        else
            begin least:= 2; second:= 1 end
        for i:= 3 to lasttree do
            if FOREST[i].weight < FOREST[least].weight then
                begin second:= least; least:= i end
            else if FOREST[i].weight < FOREST[second].weight then
                second:= i
    end; { lightones }

function create ( lefttree, righttree: integer ): integer;
    { возвращает новый узел, у которого левым и правым сыновьями
      становятся FOREST[lefttree].root и FOREST[righttree].root }
    begin
        lastnode:= lastnode + 1;
            { ячейка TREE[lastnode] для нового узла }
        TREE[lastnode].leftchild:= FOREST[lefttree].root;
        TREE[lastnode].rightchild:= FOREST[righttree].root;
        { теперь введем указатели для нового узла и его сыновей }
        TREE[lastnode].parent:= 0;
        TREE[FOREST[lefttree].root].parent:= lastnode;
        TREE[FOREST[righttree].root].parent:= lastnode;
        return(lastnode)
    end; { create }
```

Теперь все неформальные операторы листинга 3.8 можно описать подробнее. В листинге 3.10 приведен код процедуры *Huffman*, которая не осуществляет ввод и вывод, а работает со структурами, показанными на рис. 3.16, которые объявлены как глобальные.

Листинг 3.10. Реализация алгоритма Хаффмана

```
procedure Huffman;
    var
        i, j:integer; {два дерева с наименьшими весами из FOREST}
        newroot: integer;
    begin
        while lasttree > 1 do begin
            lightones(i, j);
            newroot:= created, j) ;
            { далее дерево i заменяется деревом с корнем newroot }
            FOREST[i].weight:=FOREST[i].weight + FOREST[j].weight;
            FOREST[i].root:= newroot;
            { далее дерево j заменяется на дерево lasttree,
                массив FOREST уменьшается на одну запись }
```

```
            FOREST[j] := FOREST[lasttree];
            lasttree:= lasttree - 1
        end
    end; { Huffman }
```

На рис. 3.17 показана структура данных из рис. 3.16 после того, как значение переменной *lasttree* уменьшено до 3, т.е. лес имеет вид, показанный на рис. 3.15,в.

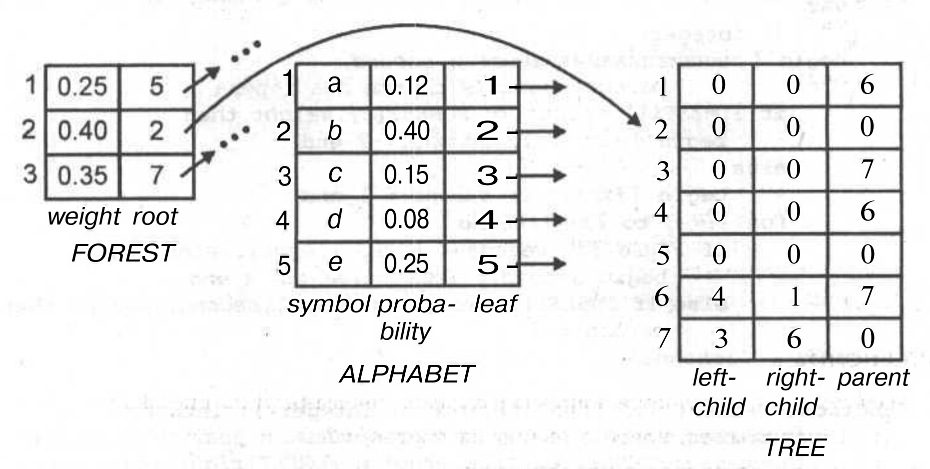


Рис. 3.17. Структура данных после двух итераций

После завершения работы алгоритма код каждого символа можно определить следующим образом. Найдем в массиве *ALPHABET* запись с нужным символ в поле *symbol.* Затем по значению поля *leaf* этой же записи определим местоположение записи в массиве ***TREE***, которая соответствует листу, помеченному рассматриваемым символом. Далее последовательно переходим по указателю *parent* от текущей записи, например соответствующей узлу *n, к* записи в массиве *TREE,* соответствующей его родителю *p.* По родителю *p* определяем, в каком его поле, ***leftchild*** или ***rightchild***, находится указатель на узел *n,* т.е. является ли узел га левым или правым сыном, и в соответствии с этим печатаем 0 (для левого сына) или 1 (для правого сына). Затем переходим к родителю узла *p* и определяем, является ли его сын *p* правым или левым, и в соответствии с эти печатаем следующую 1 или О, и т. д. до самого корня дерева. Таким образом, код символа будет напечатан в виде последовательности битов, но в обратном порядке. Чтобы распечатать эту последовательность в прямом порядке, надо каждый очередной бит помещать в стек, а затем распечатать содержимое стека в обычном порядке.

Реализация двоичных деревьев с помощью указателей

Для указания на правых и левых сыновей (и родителей, если необходимо) вместо курсоров можно использовать настоящие указатели языка Pascal. Например, можно сделать объявление

```
type
    node = record
        leftchild: T node;
        rightchild: t node;
        parent: t node
    end
```

Используя этот тип данных узлов двоичного дерева, функцию *create* (листинг 3.9) можно переписать так, как показано в следующем листинге.

Листинг 3.11. Код функции *create* при реализации двоичного дерева с помощью указателей

```
function create ( lefttree, righttree: ↑node): tnode;
    var
        root: Tnode;
    begin
        new(root);
        root↑.leftchild:= lefttree;
        root↑.rightchild:= righttree;
        root↑.parent:= 0;
        lefttreeT.parent:= root;
        righttree↑.parent:= root;
        return(root)
    end; { create }
```

Упражнения

3.1. Ответьте на следующие вопросы о дереве, показанном на рис. **3.18:**

а) какие узлы этого дерева являются листьями?

б) какой узел является корнем дерева?

в) какой узел является родителем узла С?

г) назовите сыновей узла *С;*

д) назовите предков узла *Е*;

е) назовите потомков узла *Е;*

ж) какие узлы являются правыми братьями узлов *D* и *Е?*

з) какие узлы лежат слева и справа от узла G?

и) какова глубина узла *С?*

к) какова высота узла С?

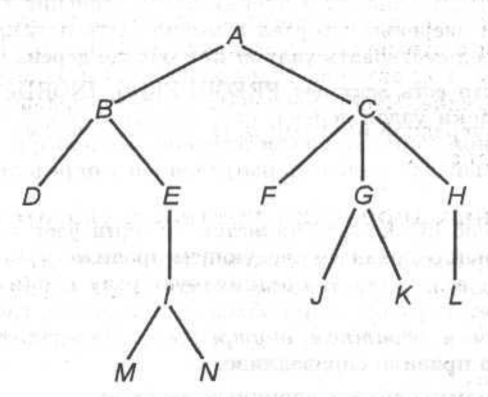


Рис. 3.18. Дерево

3.2. Сколько путей длины 3 существует на дереве, показанном на рис. 3.18?

3.3. Напишите программы вычисления высоты дерева с использованием трех представлений деревьев, описанных в разделе 3.3.

3.4. Составьте списки узлов дерева, представленного на рис. 3.18, при обходе этого дерева

а) в прямом порядке;

б) в обратном порядке;

в) во внутреннем порядке.

3.5. Пусть два различных узла *m* и *n* принадлежат одному и тому же дереву. Покажите, что только одно из следующих утверждений может быть истинным:

а) узел *m* расположен слева от узла *n*;

б) узел *m* расположен справа от узла *n*;

в) узел *m* — истинный предок узла *n*;

г) узел *m* — истинный потомок узла *n*.

3.6. Поставьте галочку в ячейку на пересечении строки *i* и столбца *j*, если одновременно выполняются условия, представленные в заголовках строки *i* и столбца *j*.

	Узел *n* предшествует узлу *m* при обходе дерева в прямом порядке	Узел *n* предшествует узлу *m* при обходе дерева в обратном порядке	Узел *n* предшествует узлу *m* при симметричном обходе дерева
Узел *n* расположен слева от узла *m*			
Узел *n* расположен справа от узла *m*			
Узел *n* — истинный предок узла *m*			
Узел *n* — истинный потомок узла *m*			

Например, поставьте галочку в ячейку на пересечение третьей строки и второго столбца, если уверены, что узел *n* может быть истинным предком узла *m* и в тоже время предшествовать узлу *m* при обходе дерева в обратном порядке.

3.7. Предположим, что есть массивы PREORDER[*n*], INORDER[*n*] и POSTORDER[n], содержащие списки узлов дерева, полученные при его обходе в прямом, внутреннем и обратном порядке соответственно. Используя эти массивы, опишите алгоритм, который для любой пары узлов *i* и *j* определяет, является ли узел i предком узла *j*.

***3.8.** Существует способ проверить, является ли один узел истинным предком другого узла, который основан на следующем правиле: узел *m* является истинным предком узла *n*, если узел *m* предшествует узлу *n* при обходе дерева в X порядке, но следует за узлом *n* при обходе в Y порядке, где X и Y выбираются из множества {*прямом, обратном, внутреннем*}. Определите все возможные пары X и Y, когда это правило справедливо.

3.9. Напишите программы обхода двоичных деревьев

а) в прямом порядке;

б) в обратном порядке;

в) во внутреннем порядке.

3.10. При прохождении дерева *в порядке уровней в* список узлов сначала заносится корень дерева, затем все узлы глубины 1, далее все узлы глубины 2 и т.д. Узлы одной глубины заносятся в список узлов в порядке слева направо. Напишите программу обхода деревьев в порядке уровней.

3.11. Преобразуйте выражение $((a + b) + c * (d + e) + f) * (g + h)$

а) в префиксную форму;

б) в постфиксную форму.

3.12. Нарисуйте дерево, соответствующее префиксным выражениям

а) $*a + b*c + de$;

б) $*a + *b + cde$.

3.13. Пусть T — дерево, в котором каждый узел, не являющийся листом, имеет ровно двух сыновей. Напишите программы преобразования

а) списка узлов дерева T, составленного при обходе дерева в прямом порядке, в список, составленный при обходе в обратном порядке;

б) списка узлов дерева T, составленного при обходе дерева в обратном порядке, в список, составленный при обходе в прямом порядке;

в) списка узлов дерева T, составленного при обходе дерева в прямом порядке, в список, составленный при симметричном обходе.

3.14. Напишите программу вычисления арифметических выражений при обходе дерева

а) в прямом порядке;

б) в обратном порядке.

3.15. Двоичное дерево можно определить как АТД со структурой двоичного дерева и операторами LEFTCHILD(n), RIGHTCHILD(n), PARENT(n) и NULL(n). Первые три оператора возвращают соответственно левого сына, правого сына и родителя узла n (если такового нет, то возвращается Л). Последний оператор возвращает значение true тогда и только тогда, когда дерево с корнем n является нулевым. Реализуйте эти операторы для двоичного дерева, представленного в табл. 3.1.

3.16. Напишите процедуры для семи операторов деревьев из раздела 3.2, используя следующие реализации деревьев:

а) указатели на родителей;

б) список сыновей;

в) указатели на самого левого сына и на правого брата.

3.17. *Степенью узла* называется количество его сыновей. Покажите, что в произвольном двоичном дереве количество листьев на единицу больше числа узлов со степенью 2.

3.18. Докажите, что в любом двоичном дереве высотой h количество узлов не превышает $2^{h+1} - 1$. Двоичное дерево высотой h с максимально возможным количеством узлов называется *полным* двоичным деревом.

*3.19. Предположим, что дерево реализуется с помощью указателей на самого левого сына, на правых братьев и на родителя. Разработайте алгоритмы обхода деревьев в прямом, обратном и внутреннем порядке, не используя при этом стека, как сделано в листинге 3.3.

3.20. Пусть символы a, b, c, d, e и f имеют вероятности появления соответственно 0.07, 0.09, 0.12, 0.22, 0.23, 0.27. Найдите оптимальный код Хаффмана и нарисуйте соответствующее ему дерево. Какова средняя длина кода?

*3.21. Пусть Г — дерево Хаффмана и пусть лист, помеченный символом a, имеет большую глубину, чем лист, помеченный символом b. Докажите, что вероятность символа b не меньше, чем вероятность символа a.

***3.22.** Докажите, что в результате выполнения алгоритма Хаффмана для заданных вероятностей получается оптимальный код. *Совет:* используйте упражнение 3.21.

Библиографические замечания

В работах [11] и [47] рассматриваются математические свойства деревьев. В [65] и [81] приведена дополнительная информация о деревьях двоичного поиска. Многие работы, приведенные в главе 6 и относящиеся к графам и их применению, содержат также материал о деревьях.

Алгоритм поиска оптимального кода, описанный в разделе 3.4, взят из работы [54]. В [83] дано современное исследование этого алгоритма.

ГЛАВА 4

Основные операторы **множеств**

Множество является той базовой структурой, которая лежит в основании всей математики. При разработке алгоритмов множества используются как основа многих важных абстрактных типов данных, и многие технические приемы и методы разработаны для реализации абстрактных типов данных, основанных именно на множествах. В этой главе мы рассмотрим основные операторы, выполняемые над множествами, и опишем некоторые простые реализации множеств. Мы представим "словарь" и "очередь с приоритетами" — два абстрактных типа данных, основанных на модели множеств. Реализации для этих АТД приведены в этой и следующей главах.

4.1. Введения в множества

Множеством называется некая совокупность *элементов,* каждый элемент множества или сам является множеством, или является примитивным элементом, называемым *атомом.* Далее будем предполагать, что все элементы любого множества различны, т.е. в любом множестве нет двух копий одного и того же элемента.

Когда множества используются в качестве инструмента при разработке алгоритмов и структур данных, то атомами обычно являются целые числа, символы, строки символов и т.п., и все элементы одного множества, как правило, имеют одинаковый тип данных. Мы часто будем предполагать, что атомы линейно упорядочены с помощью отношения, обычно обозначаемого символом "<" и читаемого как "меньше чем" или "предшествует". *Линейно упорядоченное* множество S удовлетворяет следующим двум условиям.

1. Для любых элементов a и b из множества S может быть справедливым только одно из следующих утверждений: $a < b$, $a = b$ или $b < a$.
2. Для любых элементов a, b и c из множества S таких, что $a < b$ и $b < c$, следует $a < c$ (свойство **транзитивности**)[1].

Множества целых и действительных чисел, символов и символьных строк обладают естественным линейным порядком, для проверки которого можно использовать оператор отношения < языка Pascal. Понятие линейного порядка можно определить для объектов, состоящих из множеств упорядоченных объектов. Формулировку такого определения мы оставляем читателю в качестве упражнения. (Прежде чем приступать к общей формулировке, рассмотрите пример двух множеств целых чисел, одно из которых состоит из чисел 1 и 4, а второе — из чисел 2 и 3, и сформулируйте условия, в соответствии с которыми первое множество будет больше или меньше второго.)

---

[1] Классическое определение отношения линейного строгого порядка требует выполнения свойств антирефлексивности, антисимметричности и транзитивности. Здесь выполнение свойства антирефлексивности обеспечивает требование различности элементов любого множества, а сформулированное свойство 1 является парафразом "стандартного" определения свойства антисимметричности. — *Прим. ред.*

Система обозначений для множеств

Множество обычно изображается в виде последовательности его элементов, заключенной в фигурные скобки, например {1, 4} обозначает множество, состоящее из двух элементов, — чисел 1 и 4. Порядок, в котором записаны элементы множества, не существен, поэтому мы можем писать {4, 1} так же, как и {1, 4>, при записи одного и того же множества. Отметим (и будем это иметь в виду в дальнейшем), что множество не является списком (хотя бы по признаку произвольного порядка записи элементов), но для представления множеств мы будем использовать списки. Повторим также еще раз, что мы предполагаем отсутствие повторяющихся элементов в множестве, поэтому набор элементов {1, 4, 1} не будем воспринимать как множество[1].

Иногда мы будем представлять множества с помощью *шаблонов,* т.е. выражений вида $\{x \mid$ утверждение относительно $x\}$, где "утверждение относительно x" является формальным утверждением, точно описывающим условия, при выполнении которых объект x может быть элементом множества. Например, шаблон $\{x \setminus x$ — положительное целое число и $x < 1000\}$ представляет множество {1, 2, ..., 1000}, а запись $\{x \mid$ для произвольного целого y, $x = y^2\}$ определяет множество точных квадратов целых чисел. Отметим, что множество точных квадратов бесконечно и его нельзя представить перечислением всех его элементов.

Фундаментальным понятием теории множеств является понятие отношения принадлежности к множеству, обозначаемое символом $\in$. Таким образом, запись $x \in A$ обозначает, что x принадлежит множеству A, т.е. является элементом этого множества; элемент x может быть атомом или другим множеством, но A не может быть атомом. Запись x г A означает, что "x не принадлежит множеству A", т.е. x не является элементом множества A. Существует специальное множество, обозначаемое символом 0, которое называется *пустым множеством* и которое не имеет элементов. Отметим, что 0 является именно множеством, а не атомом, хотя и не содержит никаких элементов. Утверждение x е 0 является ложным для любого элемента x, в то же время запись $x \in y$ (если x и y атомы) просто не имеет смысла: здесь синтаксическая ошибка, а не ложное утверждение.

Говорят, что множество A *содержится* в множестве B (или *включается* в множество B), пишется $A \subset B$ или $B \supseteq A$, если любой элемент множества A является также элементом множества B. В случае A с B также говорят, что множество A является *подмножеством множества B.* Например, {1, 2} с {1, 2, 3}, но множество {1, 2, 3} не может быть подмножеством множества {1, 2}, поскольку множество {1, 2, 3} содержит элемент 3, которого не имеет множество {1, 2}. Каждое множество включается в самого себя, пустое множество содержится в любом множестве. Два множества равны, если каждое из них содержится в другом, т.е. они имеют одни и те же элементы. Если $A \neq B$ и A с B, то множество A называют *собственным, истинным* или *строгим подмножеством* множества B.

Основными операциями, выполняемыми над множествами, являются операции объединения, пересечения и разности. *Объединением* множеств A и B (обозначается A и B) называется множество, состоящее из элементов, принадлежащих хотя бы одному из множеств A и B. *Пересечением* множеств A и B (обозначается $A \cap B$) называется множество, состоящее только из тех элементов, которые принадлежат и множеству A, и множеству B. *Разностью* множеств A и B (обозначается $A \setminus B$) называется множество, состоящее только из тех элементов множества A, которые не принадлежат множеству B. Например, если $A = \{a, b, c\}$ и $B = \{b, d\}$, то A и $B = \{a, b, c, d\}$, $A \cap B = \{b\}$ и $A \setminus B = \{a, c\}$.

[1] Для обозначения "множества с повторениями" иногда используется термин *мультимножество.* Мультимножеством будет набор элементов {1, 4, 1}. Мультимножество также не является списком и даже в большей степени, чем простое множество, поскольку для мультимножества можно писать {4, 1, 1} и {1, 1, 4}.

Операторы АТД, основанные на множествах

Рассмотрим операторы, выполняемые над множествами, которые часто включаются в реализацию различных абстрактных типов данных. Некоторые из этих операторов имеют общепринятые (на сегодняшний день) названия и эффективные методы их реализации. Следующий список содержит наиболее общие и часто используемые операторы множеств (т.е. выполняемые над множествами).

1-3. Первые три процедуры UNION(A, *B, C)*, INTERSECTION(*A, B, C*) и DIFFERENCE(*A, B, C*)[1] имеют "входными" аргументами множества Л и В, а в качестве результата — "выходное" множество *С*, равное соответственно $A \cup B$, $A \cap B$ и $A \setminus B$.

4. Иногда мы будем использовать оператор, который называется *слияние* (merge), или *объединение непересекающихся множеств*. Этот оператор (обозначается MERGE) не отличается от оператора объединения двух множеств, но здесь предполагается, что множества-операнды *не пересекаются* (т.е. не имеют общих элементов). Процедура MERGE(A, *B, C)* присваивает множеству *С* значение А и В, но результат будет не определен, если $A \cap B \neq 0$, т.е. в случае, когда множества А и В имеют общие элементы.

5. Функция MEMBER(*x, A)* имеет аргументами множество *А* и объект *x* того же типа, что и элементы множества *А*, и возвращает булево значение true (истина), если *x* £ *A*, и значение false (ложь), если $x \notin A$.

6. Процедура MAKENULL(*A*) присваивает множеству *А* значение пустого множества.

7. Процедура INSERT(*x, A)*, где объект *x* имеет тот же тип данных, что и элементы множества *А*, делает *x* элементом множества *А*. Другими словами, новым значением множества *А* будет $A \cup \{x\}$. Отметим, что в случае, когда элемент *x* уже присутствует в множестве *А*, это множество не изменяется в результате выполнения данной процедуры.

8. Процедура DELETE(*x*, A) удаляет элемент *x* из множества А, т.е. заменяет множество А множеством $A \setminus \{x\}$. Если элемента *x* нет в множестве А, то это множество не изменяется.

9. Процедура ASSIGN(A, B) присваивает множеству А в качестве значения множество В.

10. Функция MIN(*A*) возвращает наименьший элемент множества А. Для применения этой функции необходимо, чтобы множество А было параметризировано и его элементы были линейно упорядочены. Например, MIN({2, 3, 1}) = 1 и MIN({'*a*', '*b*', '*c*'})— '*a*'. Подобным образом определяется функция MAX.

11. Функция EQUAL(*A*, B) возвращает значение true тогда и только тогда, когда множества А и В состоят из одних и тех же элементов.

12. Функция FIND(*x*) оперирует в среде, где есть набор непересекающихся множеств. Она возвращает имя (единственное) множества, в котором есть элемент *x*.

4.2. АТД с операторами множеств

Мы начнем с определения АТД для математической модели "множество" С определенными тремя основными теоретико-множественными операторами объединения, пересечения и разности. Сначала приведем пример такого АТД и покажем, как его можно использовать, а затем обсудим некоторые простые реализации этого АТД.

---

[1] Названия процедур переводятся как "Объединение", "Пересечение" и "Разность". — *Прим. перев.*

Пример 4.1. Напишем программу, выполняющую простой "анализ потока данных" по блок-схемам представленных процедур. Программа будет использовать переменные абстрактного типа данных SET (Множество), для этого АТД операторы UNION, INTERSECTION, DIFFERENCE, EQUAL, ASSIGN и MAKENULL определены в предыдущем разделе.

В **блок-схеме** на рис. 4.1 блоки-прямоугольники помечены B_1, ..., B_8, а операторы *определения данных* (операторы объявления данных и операторы присваивания) пронумерованы от 1 до 9. Эта блок-схема соответствует алгоритму Евклида (вычисление наибольшего общего делителя двух **чисел**), но в данном примере детали этого алгоритма нас не интересуют.

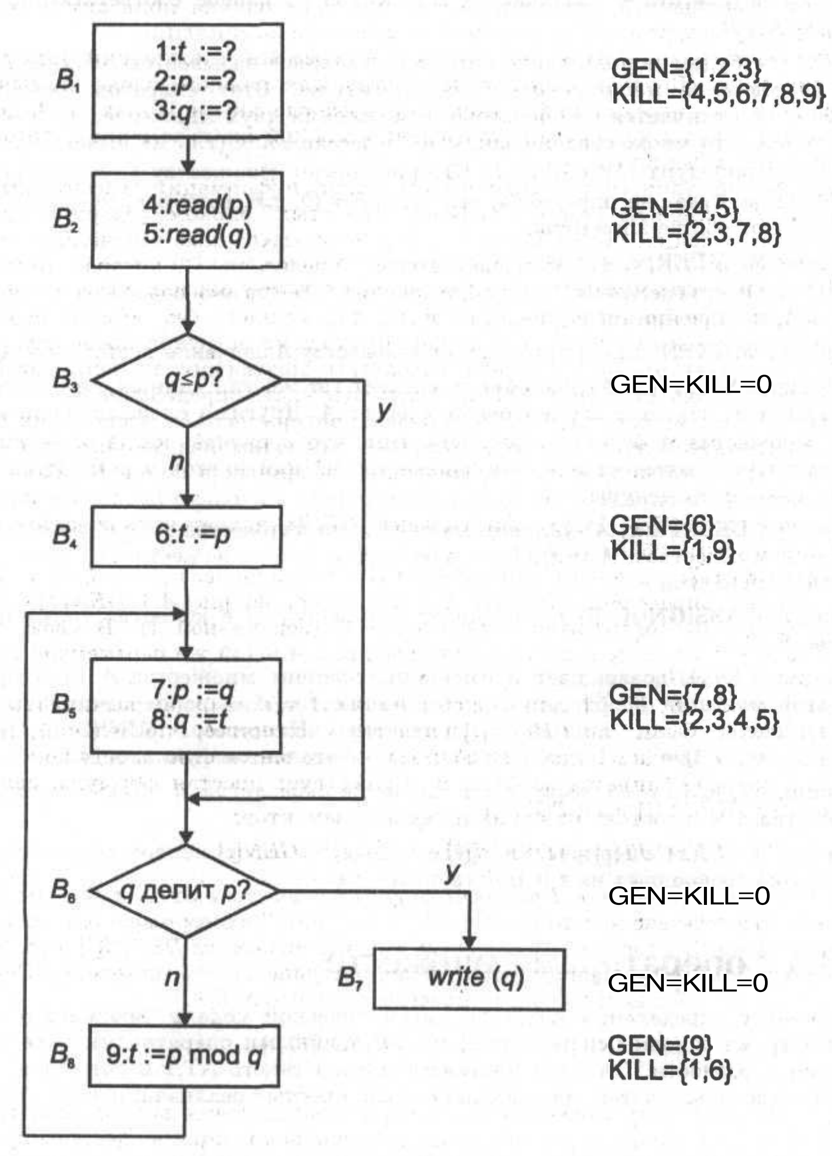


Рис. 4.1. Блок-схема алгоритма Евклида

В общем случае *анализ потока данных* относится к той части компилятора, которая проверяет блоковую структуру исходной программы (подобную рис. 4.1) и собирает информацию о выполнении операторов каждого прямоугольника блок-схемы. *Блоки* (прямоугольники) блок-схемы представляют совокупность операторов, через которые последовательно "проходит" поток данных. Информация, полученная в результате анализа потока данных, помогает повысить качество генерируемого компилятором кода программы. Если, например, в процессе анализа потока данных обнаружено, что при каждом прохождении блока B переменная x принимает значение 27, то можно подставить 27 для всех вхождений x в блоке B вместо выполнения оператора присваивания этой переменной. Если доступ к константам осуществляется быстрее, чем к значениям переменных, то описанная замена приводит к более эффективному коду программы, полученному в процессе компиляции.

В нашем примере мы хотим определить, **где** переменная последний раз принимала новое значение. Другими словами, мы хотим **для** каждого блока B_i вычислить множество *DEFIN[i]* определений d таких, которые сделаны в блоках от B_1 до B_i, но в последующих блоках нет других определений для той же переменной, которая определялась в определении d.

Чтобы проиллюстрировать необходимость такой информации, рассмотрим блок-схему на рис. 4.1. Первый блок B_1, являясь "холостым" блоком, содержит три определения, присваивающих переменным t, p и q "неопределенные" значения. Если, например, множество *DEFIN[7]* включает в себя определение 3, которое присваивает переменной q "неопределенное" значение, значит, программа содержит ошибку, поскольку будет предпринята печать этой переменной без присваивания ей "настоящего" значения. К счастью, в данной блок-схеме нетрудно проверить, что невозможно достигнуть блока B_7, минуя операторы присвоения значений этой переменной, поэтому множество *DEFIN*[7] не будет содержать определение 3.

При вычислении множества *DEFIN*[i] будем придерживаться нескольких правил. Сначала для каждого блока B_i предварительно вычислим два множества *GEN[i]* и *KILL*[i]. *GEN*[i] — это множество определений, сделанных в блоке Д. Если в этом блоке есть несколько определений одной переменной, то только последнее из них заносится в множество *GEN[i]*. Другими словами, *GEN[i]* является множеством определений, "генерируемых" блоком Д.

Множество *KILL*[i] содержит определения (из всех блоков, кроме Д) тех же переменных, которые определены в блоке B_i. Например, на рис. 4.1 *GEN*[4] — {6}, поскольку в блоке B_4 содержится определение 6 (переменной t). В свою очередь *KILL*[4] = {1, 9}, так как существуют определения 1 и 9 той же переменной t и которые не входят в блок B_4.

Кроме множеств *DEFIN*[i], для каждого блока Д также будем вычислять множества *DEFOUT*[i]. Так же, как *DEFIN*[i] является множеством определений, действие которых достигает блока Д, так и *DEFOUT*[i] — это множество определений, действие которых распространяется за блок B_i. Существует простая формула, связывающая множества *DEFIN[i]* и *DEFOUT*[i]:

$$DEFOUT[i] = (DEFIN[i] - KILL[i]) \cup GEN[i] \qquad \textbf{(4.1)}$$

Таким образом, определение d распространяет свое действие за блок B_i только в двух случаях: если его действие начато до блока Д и не "убито" в этом блоке или если оно генерировано в блоке B_i. Вторая формула, связывающая множества *DEFIN*[i] и *DEFOUT*[i], показывает, что *DEFIN*[i] можно вычислить как объединение тех множеств *DEFOUT*[p], для которых определяющие их блоки B_p предшествуют блоку B_i.

$$DEFIN[i] = \bigcup_{B_p \text{ предшествует Д}} DEFOUT[p] \qquad (4.2)$$

Из формулы (4.2) следует очевидный вывод, что определения данных регистрируются в блоке B_i тогда и только тогда, когда их действие начинается в одном из предшествующих блоков. В особых случаях, когда Д не имеет предшественников (как блок B_1 на рис. 4.1), множество *DEFIN*[i] считается равным пустому множеству.

Хотя в этом примере введено много новых понятий, мы не будем обобщать этот материал для написания общих алгоритмов вычисления областей действия определений в произвольных блок-схемах. Вместо этого мы напишем часть программы вычисления множеств *DEFIN[i]* и *DEFOUT[i]* (i = 1, ..., 8) для блок-схемы рис. 4.1, предполагая, что множества *GEN[i]* и *KILL[i]* определены заранее. Отметим, что предлагаемый фрагмент программы предполагает существование АТД SET (Множество) с операторами UNION, INTERSECTION, DIFFERENCE, EQUAL, ASSIGN и MAKENULL, позднее мы покажем, как реализовать этот АТД.

В создаваемом программном фрагменте (листинг 4.1) процедура *propagate(GEN, KILL, DEFIN, DEFOUT)* использует формулу (4.1) для вычисления множества *DEFOUT* указанного блока. Если программа свободна от циклов и повторяющихся структур, то в этом случае множества *DEFOUT* можно вычислить непосредственно. При наличии в программе циклов или повторяющихся структур для вычисления *DEFOUT* применяются итерационные процедуры. Сначала для всех i положим *DEFIN[i]* = 0 и *DEFOUT[i]* = *GEN[i]*, затем последовательно применим формулы (4.1) и (4.2) несколько раз, пока не "стабилизируются" (не перестанут изменяться) множества *DEFIN* и *DEFOUT*. Поскольку каждое новое значение множеств *DEFIN[i]* и *DEFOUT[i]* содержит свои предыдущие значения, а количество всех определений в программе ограничено, то процесс должен сходиться к решению уравнений (4.1) и (4.2).

Последовательные значения множеств *DEFIN[i]* после каждой итерации цикла показаны в табл. 4.1. Заметим, что операторы 1, 2 и 3 присваивания переменным "неопределенных" значений не распространяются на блоки, где эти переменные используются. Отметим также, что в листинге 4.2 сначала выполняются вычисления по формуле (4.2), а затем по формуле (4.1); в общем случае это обеспечивает сходимость процесса вычисления *DEFIN[i]* всего за несколько итераций. □

Листинг 4.1. Программа вычисления областей действий операторов определения переменных

```
var
    GEN, KILL, DEFIN, DEFOUT: array[1..8] of SET;
        { Предполагается, что GEN и KILL вычисляются
          вне этой программы }
    i: integer;
    changed: boolean;

procedure propagate( G, K, I: SET; var O: SET ) ;
    {- Вычисления по формуле (4.1), переменная changed принимает
        значение true, если есть изменения в DEFOUT }
    var
        TEMP: SET;
    begin
        DIFFERENCE(I, K, TEMP) ;
        UNION ( TEMP, G, TEMP);
        if not EQUAL (TEMP, O) do begin
            ASSIGN(O, TEMP);
            changed:= true
        end
    end; { propagate }

begin
    for i:= 1 to 8 do
        ASSIGN(DEFOUT[i] , GEN[i]);
    repeat
```

```
          changed:= false;
          { далее 8 операторов реализуют формулу (4.2), но только
               для блок-схемы рис. 4.1 }
          MAKENULL(DEFIN[1]);
          ASSIGN(DEFIN[2], DEFOUT[1]);
          ASSIGN(DEFIN[3], DEFOUT[2]);
          ASSIGN(DEFIN[4], DEFOUT[3]);
          UNION(DEFOUT[4], DEFOUT[8], DEFIN[5]);
          UNION(DEFOUT[3], DEFOUT[5], DEFIN[6]);
          ASSIGN(DEFIN[7], DEFOUT[6]);
          ASSIGN(DEFIN[8], DEFOUT[6]);
          for i:= 1 to 8 do
               propagate(GEN[i], KILL[i], DEFIN[i], DEFOUT[i]);
     until
          not changed
end.
```

Таблица 4.1. Значения *DEFIN[i]* после каждой итерации

i	Проход 1	Проход 2	Проход 3	Проход 4
1	∅	∅	∅	∅
2	{1, 2, 3}	{1, 2, 3}	{1, 2, 3}	{1, 2, 3}
3	{4, 5}	{1, 4, 5}	{1, 4, 5}	{1, 4, 5}
4	∅	{4, 5}	{1, 4, 5}	{1, 4, 5}
5	{6, 9}	{6, 9}	{4, 5, 6, 7, 8, 9}	{4, 5, 6, 7, 8, 9}
6	{7, 8}	{4, 5, 6, 7, 8, 9}	{1, 4, 5, 6, 7, 8, 9}	{1, 4, 5, 6, 7, 8, 9}
7	∅	{7, 8}	{4, 5, 6, 7, 8, 9}	{1, 4, 5, 6, 7, 8, 9}
8	∅	{7, 8}	{4, 5, 6, 7, 8, 9}	{1, 4, 5, 6, 7, 8, 9}

4.3. Реализация множеств посредством двоичных векторов

Наилучшая конкретная реализация абстрактного типа данных SET (Множество) выбирается исходя из набора выполняемых операторов и размера множества. Если все рассматриваемые множества будут подмножествами небольшого *универсального множества* целых чисел 1, ..., N для некоторого фиксированного N, тогда можно применить реализацию АТД SET посредством двоичного (булева) вектора. В этой реализации множество представляется двоичным вектором, в котором i-й бит равен 1 (или true), если i является элементом множества. Главное преимущество этой реализации состоит в том, что здесь операторы MEMBER, INSERT и DELETE можно выполнить за фиксированное время (независимо от размера множества) путем прямой адресации к соответствующему биту. Но операторы UNION, INTERSECTION и DIFFERENCE выполняются за время, пропорциональное размеру универсального множества.

Если универсальное множество так мало, что двоичный вектор занимает не более одного машинного слова, то операторы UNION, INTERSECTION и DIFFERENCE можно выполнить с помощью простых логических операций (конъюнкции и дизъюнкции), выполняемых над двоичными векторами. В языке Pascal для представления небольших множеств можно использовать встроенный тип данных set. Максимальный размер таких множеств зависит от конкретного применяемого компилятора

и поэтому не формализуем. Однако при написании программ мы не хотим быть связаны ограничением максимального размера множеств по крайней мере до тех пор, пока наши множества можно трактовать как подмножества некоего универсального множества {1, ..., N}. Поэтому в данном разделе будем придерживаться представления множества в виде булевого массива A, где $A[i]$ = true тогда и только тогда, когда i является элементом множества. С помощью объявлений языка Pascal АТД SET можно определить следующим образом:

```
const
    N = { подходящее числовое значение };
type
    SET = packed array[1..N] of boolean;
```

Реализация оператора UNION приведена в листинге 4.2. Для реализации операторов INTERSECTION и DIFFERENCE надо в листинге 4.2 заменить логический оператор or на операторы and и and not соответственно. В качестве простого упражнения читатель может самостоятельно написать код для реализации других операторов, перечисленных в разделе 4.1 (за исключением операторов MERGE и FIND, которые не имеют практического смысла при данном представлении множеств).

Листинг 4.2. Реализация оператора UNION

```
procedure UNION ( A, B: SET; var C: SET );
    var
        i: integer;
    begin
        for i:= 1 to N do
            C[i]:= A[i] or B[i]
    end
```

Представление множеств в виде двоичных векторов можно применить не только тогда, когда универсальное множество состоит из последовательных целых чисел, но и в более общей ситуации, когда универсальное множество конечно. Для этого достаточно установить взаимно однозначное соответствие между элементами этого множества и целыми числами 1, ..., N. Например, так, как это сделано в примере 4.1, где все определения данных мы пронумеровали числами от 1 до 9. В общем случае для реализации такого отображения используется АТД MAPPING (Отображение), описанный в главе 2. На практике обратное отображение из множества целых чисел в множество элементов универсального множества проще выполнить с помощью массива A, где $A[i]$ будет элементом универсального множества, соответствующим числу i.

4.4. Реализация множеств посредством связанных списков

Очевиден способ представления множеств посредством связанных списков, когда элементы списка являются элементами множества. В отличие от представления множеств посредством двоичных векторов, в данном представлении занимаемое множеством пространство пропорционально размеру представляемого множества, а не размеру универсального множества. Кроме того, представление посредством связанных списков является более общим, поскольку здесь множества не обязаны быть подмножествами некоторого конечного универсального множества.

При реализации оператора INTERSECTION (Пересечение) в рамках представления множеств посредством связанных списков есть несколько альтернатив. Если универсальное множество линейно упорядочено, то в этом случае множество можно представить в виде сортированного списка, т.е. предполагая, что все элементы множества сравнимы посредством отношения "<", можно ожидать, что эти элементы в списке

будут находиться в порядке $e_1, e_2, ..., e_n$, когда $e_1 < e_2 < e_3 < ... < e_n$. Преимущество отсортированного списка заключается в том, что для нахождения конкретного элемента в списке нет необходимости просматривать весь список.

Элемент будет принадлежать пересечению списков L_1 и L_2 тогда и только тогда, когда он содержится в обоих списках. В случае несортированных списков мы должны сравнить каждый элемент списка L_1 с каждым элементом списка L_2, т.е. сделать порядка $O(n^2)$ операций при работе со списками длины п. Для сортированных списков операторы пересечения и некоторые другие выполняются сравнительно просто: если надо сравнить элемент e списка L_1 с элементами списка L_2, то надо просматривать список L_2 только до тех пор, пока не встретится элемент e или больший, чем e. В первом случае будет совпадение элементов, второй случай показывает, что элемента e нет в списке L_2. Более интересна ситуация, когда мы знаем элемент d, который в списке L_1 непосредственно предшествует элементу e. Тогда для поиска элемента, совпадающего с элементом e, в списке L_2 можно сначала найти такой элемент f, что $d \leq f$, и начать просмотр списка L_2 с этого элемента. Используя этот прием, можно найти совпадающие элементы в списках L_1 и L_2 за один проход этих списков, продвигаясь вперед по спискам в прямом порядке, начиная с наименьшего элемента. Код процедуры, реализующей оператор INTERSECTION, показан в листинге 4.3. Здесь множества представлены связанными списками ячеек, чей тип определяется следующим образом:

```
type
    celltype = record
        element: elementtype;
        next: ↑celltype
    end
```

В листинге 4.3 предполагается, что elementtype — это тип целых чисел, которые можно упорядочить посредством обычного оператора сравнения <. Если elementtype представлен другим типом данных, то надо написать функцию, которая будет определять, какой из двух заданных элементов предшествует другому элементу.

Листинг 4.3. Процедура INTERSECTION, использующая связанные списки

```
       procedure INTERSECTION ( ahead, bhead: ↑celltype;
                   var pc: ↑celltype );
           { Вычисление пересечения сортированных списков А и В с
               ячейками заголовков ahead и bhead, результат —
               сортированный список, на чей заголовок указывает pc }
       var
           acurrent, bcurrent, ccurrent: ↑celltype;
           { текущие ячейки списков А и В и последняя ячейка
               дополнительного списка С }
       begin
(1)        new(pc); { создание заголовка для списка С }
(2)        acurrent:= ahead↑.next;
(3)        bcurrent:= bhead↑.next;
(4)        ccurrent:= pc;
(5)        while (acurrent <> nil) and (bcurrent <> nil) do begin
               { сравнение текущих элементов списков А и В }
(6)            if acurrent↑.element = bcurrent↑.element then begin
                   { добавление элемента в пересечение }
(7)                new(ccurrent↑.next);
(8)                ccurrent:= ccurrent↑.next;
(9)                ccurrent↑.element:= acurrent↑.element;
(10)           acurrent:= acurrent↑.next;
```

```
(11)            bcurrent:= bcurrent↑.next
                end
                else { элементы неравны }
(12)            if acurrent↑.element < bcurrent↑.element then
(13)                acurrent:= acurrent↑.next;
                else
(14)                bcurrent:= bcurrent↑.next
            end
(15)        ccurrent↑.next:= nil
    end; { INTERSECTION }
```

Связанные списки в листинге 4.3 в качестве заголовков имеют пустые ячейки, которые служат как указатели входа списков. Читатель при желании может написать эту программу в более общей абстрактной форме с использованием примитивов списков. Но программа листинга 4.3 может быть более эффективной, чем абстрактная программа. Например, в листинге 4.3 используются указатели на отдельные ячейки вместо "позиционных" переменных, указывающих на предыдущую ячейку. Так можно сделать вследствие того, что элементы добавляются в список С, а списки А и В только просматриваются без вставки или удаления в них элементов.

Процедуру INTERSECTION листинга 4.3 можно легко приспособить для реализации операторов UNION и DIFFERENCE. Для выполнения оператора UNION надо все элементы из списков *А и В* записать в список *С.* Поэтому, когда элементы не равны (строки 12-14 в листинге 4.3), наименьший из них заносится в список С, так же, как и в случае их равенства. Элементы заносятся в список С до тех пор, пока на исчерпаются оба списка, т.е. пока логическое выражение в строке *5* не примет значение false. В процедуре DIFFERENCE в случае равенства элементов они не заносятся в список *С.* Если текущий элемент списка А меньше текущего элемента списка *В,* то он (текущий элемент списка *А)* заносится в список С. Элементы заносятся в список С до тех пор, пока не исчерпается список *А* (логическое условие в строке *5).*

Оператор ASSIGN(A, *В)* копирует список А в список *В.* Отметим, что этот оператор нельзя реализовать простым переопределением заголовка списка *В* на заголовок списка А, поскольку при последующих изменениях в списке *В* надо будет делать аналогичные изменения в списке А, что, естественно, может привести к нежелательным коллизиям. Оператор MIN реализуется легко — просто возвращается первый элемент списка. Операторы DELETE и FIND можно реализовать, применив общие методы поиска заданного элемента в списках, в случае оператора DELETE найденный элемент (точнее, ячейка, в которой он находится) удаляется.

Реализовать оператор вставки нового элемента в список также несложно, но он должен стоять не в произвольной позиции в списке, а в "правильной" позиции, учитывающей взаимный порядок элементов. В листинге 4.4 представлен код процедуры INSERT (Вставка), которая в качестве параметров имеет вставляемый элемент и указатель на ячейку заголовка списка, куда вставляется элемент. На рис. 4.2 показаны ключевые ячейки и указатели до и после вставки элемента (старые указатели обозначены сплошными линиями, а новые — пунктирными).

Листинг 4.4. Процедура вставки элемента

```
procedure INSERT ( x: elementtype; p: tcelltype );
    var
        current, newcell: Tcelltype;
    begin
        current:= p;
        while current↑.next <> nil do begin
            if current↑.next↑.element = x then
                return; { элемент x уже есть в списке }
            if current↑.next↑.element > x then
```

```
          goto add; { далее останов процедуры }
        current:= current↑.next
    end;
add: {здесь current — ячейка, после которой надо вставить x}
    new(newcell);
    newcell↑.element:= x;
    newcell↑.next:= current↑.next;
    current↑.next:= newcell
end; { INSERT }
```

Рис. 4.2. Схема вставки нового элемента

4.5. Словари

Применение множеств при разработке алгоритмов не всегда требует таких мощных операторов, как операторы объединения и пересечения. Часто достаточно только хранить в множестве "текущие" объекты с периодической вставкой или удалением некоторых из них. Время от времени также возникает необходимость узнать, присутствует ли конкретный элемент в данном множестве. Абстрактный тип множеств с операторами INSERT, DELETE и MEMBER называется DICTIONARY (Словарь). Мы также включим оператор MAKENULL в набор операторов словаря — он потребуется при реализации АТД для инициализации структур данных. Далее в этом разделе мы приведем пример использования словарей, а в следующем разделе рассмотрим реализации, подходящие для представления словарей.

Пример 4.2. Общество защиты тунцов (ОЗТ) имеет базу данных с записями результатов самого последнего голосования законодателей по законопроектам об охране тунцов. База данных состоит из двух списков (множеств) имен законодателей, которые названы *goodguys* (хорошие парни) и *badguys* (плохие парни). ОЗТ прощает законодателям их прошлые "ошибки", но имеет тенденцию забывать своих "друзей", которые ранее голосовали "правильно". Например, после голосования по законопроекту об ограничении вылова тунца в озере Эри все законодатели, проголосовавшие за этот законопроект, заносятся в список *goodguys* и удаляются из списка *badguys*, тогда как над оппонентами этого законопроекта совершается обратная процедура. За-

конодатели, не принимавшие участие в голосовании, остаются в тех списках, в которых они были ранее.

Для управления описываемой базы данных при вводе имен законодателей будем применять односимвольные команды, за символом команды будет следовать 10 символов с именем законодателя. Каждая команда располагается в отдельной строке. Используем следующие односимвольные команды.

1. F (законодатель голосовал "правильно").
2. U (законодатель голосовал "неправильно").
3. ? (надо определить статус законодателя).

Мы также будем использовать символ 'E' для обозначения окончания процесса ввода списка законодателей. В листинге 4.5 показан эскиз программы *tuna* (тунец), написанный в терминах пока не определенного АТД DICTIONARY (Словарь), который в данном случае можно представить как множество символьных строк длиной 10. □

Листинг 4.8. Программа управления базой данных ОЗТ

```
program tuna ( input, output );
    { База данных законодателей (legislator) }
    type
        nametype = array[1..10] of char;
    var
        command: char;
        legislator: nametype;
        goodguys, badguys: DICTIONARY;

    procedure favor ( friend: nametype );
        { заносит имя friend (друг) в список goodguys
            и вычеркивает из списка badguys }
        begin
            INSERT(friend, goodguys);
            DELETE(friend, badguys)
        end; { favor }

    procedure unfavor ( foe: nametype );
        { заносит имя foe (враг) в список badguys
            и вычеркивает из списка goodguys }
        begin
            INSERT(foe, badguys);
            DELETE(foe, goodguys)
        end; { unfavor }

    procedure report ( subject: nametype );
        {печать имени subject с соответствующей характеристикой}
        begin
            if MEMBER(subject, goodguys) then
                writeln(subject, ' — это друг')
            else if MEMBER(subject, badguys) then
                writeln(subject, ' — это враг')
            else
                writeln('Нет данных о ', subject,)
        end; { report }

begin { основная программа }
    MAKENULL(goodguys);
```

```
        MAKENULL(badguys);
        read(command);
        while command <> 'E' do begin
            readln(legislator);
            if command = 'F' then
                favor(legislator)
            else if command = 'U' then
                unfavor(legislator)
            else if command = '?' then
                report(legislator)
            else
                report('Неизвестная команда')
            read(command)
        end
    end; { tuna }
```

4.6. Реализации словарей

Словари можно представить посредством сортированных или несортированных связанных списков. Другая возможная реализация словарей использует двоичные векторы, предполагая, что элементы данного множества являются целыми числами 1, ..., N для некоторого N или элементы множества можно сопоставить с таким множеством целых чисел.

Третья возможная реализация словарей использует массив фиксированной длины с указателем на последнюю заполненную ячейку этого массива. Эта реализация выполнима, если мы точно знаем, что размер множества не превысит заданную длину массива. Эта реализация проще реализации посредством связанных списков, но имеет следующие недостатки: множества могут расти только до определенной фиксированной величины; медленно выполняются операции удаления элементов из множества (так как требуется перемещение оставшихся элементов массива) и невозможность эффективно организовать пространство массивов (особенно если множества имеют различные размеры).

Так как мы рассматриваем реализации именно словарей (и вследствие последнего приведенного недостатка), то не будем затрагивать возможности выполнения в реализации посредством массивов операций объединения и пересечения множеств. Вместе с тем, поскольку массивы, так же, как и списки, можно сортировать, то читатель вправе рассматривать реализацию с помощью массивов (которую мы здесь применяем только для представления словарей) в качестве приемлемой реализации множеств произвольной структуры. В листинге 4.6 приведены объявления и процедуры, являющиеся необходимым дополнением программы листинга 4.5, — вместе с этими дополнениями программа должна работать.

Листинг 4.6. Объявления типов и процедуры реализации словаря посредством массива

```
    const
        maxsize = { некое число, максимальный размер массива }
    type
        DICTIONARY = record
            last: integer;
            data: array[1..maxsize] of nametype
        end;

    procedure MAKENULL ( var A: DICTIONARY );
        begin
```

```
      A.last:= 0
   end; { MAKENULL }

function MEMBER ( x: nametype; var A: DICTIONARY ): boolean;
   var
      i: integer;
   begin
      for i:= 1 to A.last do
         if A.data[i] - x then return(true);
      return(fale) { элемент x не найден }
   end; { MEMBER }

procedure INSERT ( x: nametype; var A: DICTIONARY );
   begin
      if not MEMBER(x, A) then
         if A.last < maxsize then begin
            A.last:= A.last + 1;
            A.data[A.last]:= x
         end
         else error('База данных заполнена')
   end; { INSERT }

procedure DELETE ( x: nametype; var A: DICTIONARY );
   var
      i:= integer;
   begin
      if A.last > 0 then begin
         i:= 1;
         while (A.data[i] <> x) and (i < A.last) do
            i:= i + 1;
         if A.data[i] = x then begin
            A.data[i] = A.data[A.last];
                 { перемещение последнего элемента на место
                   элемента x; если i = A.last, то удаление x
                   происходит на следующем шаге }
            A.last:= A.last - I
         end
      end
   end; { DELETE }
```

4.7. Структуры данных, основанные на хеш-таблицах

В реализации словарей с помощью массивов выполнение операторов INSERT, DELETE и MEMBER требует в среднем $O(N)$ выполнений элементарных инструкций для словаря из N элементов. Подобной скоростью выполнения операторов обладает и реализация с помощью списков. При реализации словарей посредством двоичных векторов все эти три оператора выполняются за фиксированное время независимо от размера множеств, но в этом случае мы ограничены множествами целых чисел из некоторого небольшого конечного интервала.

Существует еще один полезный и широко используемый метод реализации словарей, который называется *хешированием*. Этот метод требует фиксированного времени (в среднем) на выполнение операторов и снимает ограничение, что множества должны быть подмножествами некоторого конечного универсального множества. В самом худшем случае этот метод для выполнения операторов требует времени, пропорцио-

нального размеру множества, так же, как и в случаях реализаций посредством массивов и списков. Но при тщательной разработке алгоритмов мы можем сделать так, что вероятность выполнения операторов за время, большее фиксированного, будет как угодно малой.

Мы рассмотрим две различные формы хеширования. Одна из них называется *открытым,* или *внешним, хешированием* и позволяет хранить множества в потенциально бесконечном пространстве, снимая тем самым ограничения на размер множеств. Другая называется *закрытым,* или *внутренним, хешированием*[1] и использует ограниченное пространство для хранения данных, ограничивая таким образом размер множеств[2].

Открытое хеширование

На рис. 4.3 показана базовая структура данных при открытом хешировании. Основная идея заключается в том, что потенциальное множество (возможно, бесконечное) разбивается на конечное число классов. Для B классов, пронумерованных от О до B - 1, строится *хеш-функция* h такая, что для любого элемента x исходного множества функция $h(x)$ принимает целочисленное значение из интервала 0, ..., B — 1, которое, естественно, соответствует классу, которому принадлежит элемент x. Элемент x часто называют *ключом,* $h(x)$ — *хеш-значением* x*,* а "классы" — *сегментами.* Мы будем говорить, что элемент x принадлежит сегменту $h(x)$.

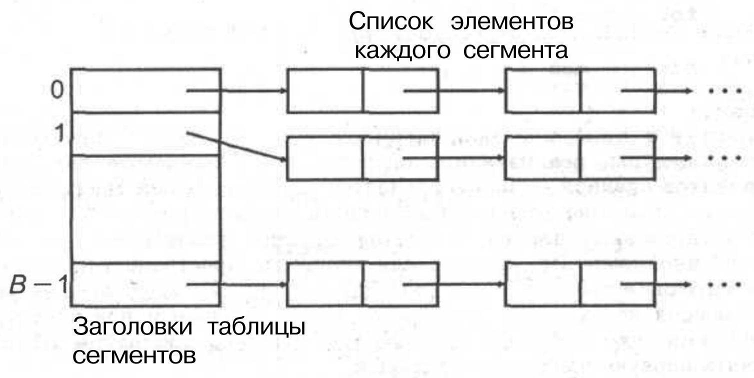


Рис. 4.3. Организация данных при открытом хешировании

Массив, называемый *таблицей сегментов* и проиндексированный *номерами сегментов* О, 1,...,В - 1, содержит заголовки для B списков. Элемент x i-го списка — это элемент исходного множества, для которого $h(x) = i$.

Если сегменты примерно одинаковы по размеру, то в этом случае списки всех сегментов должны быть наиболее короткими при данном числе сегментов. Если исходное множество состоит из N элементов, тогда средняя длина списков будет N/B элементов. Если можно оценить величину N и выбрать B как можно ближе к этой величине, то в каждом списке будет один-два элемента. Тогда время выполнения операторов словарей будет малой постоянной величиной, зависящей от N (или, что эквивалентно, от В).

---

[1] Есть несколько классификаций методов хеширования по разным признакам. Мы оставляем здесь классификацию (и терминологию) авторов, поскольку она проста и непротиворечива. Отметим только, что открытое хеширование, как оно изложено далее, является частным случаем так называемого *расширенного хеширования,* а закрытое хеширование часто называют *прямым хешированием. — Прим. ред.*

[2] Путем внесения изменений в структуру данных можно изменять объем пространства, занимаемого данными при открытом хеширование, и увеличить его при закрытом хешировании. Мы опишем эти технологии после знакомства с основными методами хеширования.

Не всегда ясно, как выбрать хеш-функцию А так, чтобы она примерно поровну распределяла элементы исходного множества по всем сегментам. Ниже мы покажем простой способ построения функции А, причем *h(x)* будет "случайным" значением, почти независящим от *x*.

Здесь же мы введем хеш-функцию (которая будет "хорошей", но не "отличной"), определенную на символьных строках. Идея построения этой функции заключается в том, чтобы представить символы в виде целых чисел, используя для этого машинные коды символов. В языке Pascal есть встроенная функция **ord(c)**, которая возвращает целочисленный код символа *c*. Таким образом, если *x* — это ключ, тип данных ключей определен как **array[1..10]** of char (в примере 4.2 этот тип данных назван **nametype**), тогда можно использовать хеш-функцию, код которой представлен в листинге 4.7. В этой функции суммируются все целочисленные коды символов, результат суммирования делится на *B* и берется остаток от деления, который будет целым числом из интервала от 0 до *B* - 1.

Листинг 4.7. Простая хеш-функция

```
function h ( x: nametype ): 0..B-1;
    var
        i, sum: integer;
    begin
        sum:= 0;
        for i:= 1 to 10 do
            sum:= sum + ord(x[i]);
        h:= sum mod B
    end; { h }
```

В листинге 4.8 показаны объявления структур данных для открытой хеш-таблицы и процедуры, реализующие операторы, выполняемые над словарем. Тип данных элементов словаря — nametype (здесь — символьный массив), поэтому данные объявления можно непосредственно использовать в примере 4.2. Отметим, что в листинге 4.8 заголовки списков сегментов сделаны указателями на ячейки, а не "настоящими" ячейками. Это сделано для экономии пространства, занимаемого данными: если заголовки таблицы сегментов будут ячейками массива, а не указателями, то под этот массив необходимо столько же места, сколько и под списки элементов. Но за такую экономию пространства надо платить: код процедуры DELETE должен уметь отличать первую ячейку от остальных.

Листинг 4.8. Реализация словарей посредством открытой хеш-таблицы

```
const
    B = { подходящая константа };
type
    celltype = record
        element: nametype;
        next: ↑celltype
    end;
    DICTIONARY = array[0..B-1] of ↑celltype;

procedure MAKENULL ( var A: DICTIONARY );
    var
        i:= integer;
    begin
        for i:= 0 to B - 1 do
            A[i]:= nil
    end; { MAKENULL }
```

```
function MEMBER ( x: nametype; var A: DICTIONARY ): boolean;
    var
        current: tcelltype;
    begin
        current:= A[h(x)];
        { начальное значение current равно заголовку сегмента,
            которому принадлежит элемент x }
        while current o nil do
            if current↑.element = x then
                return(true)
            else
                current:= current↑.next;
        return(false) { элемент x не найден }
    end; { MEMBER }

procedure INSERT ( x: nametype; var A: DICTIONARY );
    var
        bucket: integer; { для номера сегмента }
        oldheader: tcelltype;
    begin
        if not MEMBER(x, A) then begin
            bucket:= h(x);
            oldheader: = A[bucket];
            new(A[bucket]);
            A[bucket]↑.element:= x;
            A[bucket]↑.next:= oldheader
        end
    end; { INSERT }

procedure DELETE ( x: nametype; var A: DICTIONARY );
var
    bucket: integer;
    current: ↑celltype;
begin
    bucket:= h(x);
    if A[bucket] <> nil then begin
        if A [bucket]↑.element = x then { x в первой ячейке }
            A[bucket]:= A[bucket]↑.next { удаление x из списка }
        else begin { x находится не в первой ячейке }
            current:= A[bucket];
                { current указывает на предыдущую ячейку }
            while current↑.next <> nil do
                if current↑.next↑.element = x then begin
                    current↑.next:= current↑.next↑.next;
                        { удаление x из списка }
                    return { останов }
                end
                else { x пока не найден }
                    current:= current↑.next
        end
    end
end; { DELETE }
```

Закрытое хеширование

При закрытом хешировании в таблице сегментов хранятся непосредственно элементы словаря, а не заголовки списков. Поэтому в каждом сегменте может храниться только один элемент словаря. При закрытом хешировании применяется методика *повторного хеширования*. Если мы попытаемся поместить элемент x в сегмент с номером $h(x)$, который уже занят другим элементом (такая ситуация называется *коллизией*), то в соответствии с методикой повторного хеширования выбирается последовательность других номеров сегментов $h_1(x)$, $h_2(x)$, ..., куда можно поместить элемент x. Каждое из этих местоположений последовательно проверяется, пока не будет найдено свободное. Если свободных сегментов нет, то, следовательно, таблица заполнена и элемент x вставить нельзя.

Пример 4.3. Предположим, что $B = 8$ и ключи a, Ъ, c и d имеют хеш-значения $h(a) = 3$, $h(b) = 0$, $h(c) = 4$ и $h(d) = 3$. Применим простую методику, которая называется *линейным хешированием*. При линейном хешировании $h_i(x)$ — $(h(x) + i) \bmod B$. Например, если мы хотим вставить элемент d, a сегмент 3 уже занят, то можно проверить на занятость сегменты 4, 5, 6, 7, О, 1 и 2 (именно в таком порядке).

Мы предполагаем, что вначале вся хеш-таблица пуста, т.е. в каждый сегмент помещено специальное значение *empty* (пустой), которое не совпадает ни с одним элементом словаря[1]. Теперь последовательно вставим элементы a, Ь, c и d в пустую таблицу: элемент a попадет в сегмент 3, элемент Ь — в сегмент О, а элемент c — в сегмент 4. Для элемента d $h(d)$ — 3, но сегмент 3 уже занят. Применяем функцию h_1: $h_1(d) = 4$, но сегмент 4 также занят. Далее применяем функцию h_2: $h_2(d) = 5$, сегмент 5 свободен, помещаем туда элемент d. Результат заполнения хеш-таблицы показан на рис. 4.4. □

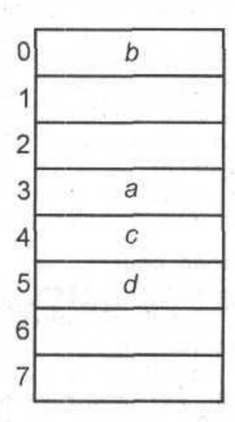


Рис. 4.4. Частично заполненная хеш-таблица

При поиске элемента x (например, при выполнении оператора MEMBER) необходимо просмотреть все местоположения $h(x)$, $h_1(x)$, $h_2(x)$, ..., пока не будет найден x или пока не встретится пустой сегмент. Чтобы объяснить, почему можно остановить поиск при достижении пустого сегмента, предположим, что в словаре не допускается удаление элементов. И пусть, для определенности, $h_3(x)$ — первый пустой сегмент. В такой ситуации невозможно нахождение элемента x в сегментах $h_4(x)$, $h_5(x)$ и далее,

[1] Если тип данных элементов словаря не соответствует типу значения *empty*, то в каждый сегмент можно поместить дополнительное однобитовое поле с маркером, показывающим, занят или нет данный сегмент.

так как при вставке элемент x вставляется в *первый* пустой сегмент, следовательно, он находится где-то до сегмента $h_3(x)$.

Но если в словаре допускается удаление элементов, то при достижении пустого сегмента мы, не найдя элемента x, не можем быть уверенными в'том, что его вообще нет в словаре, так как сегмент может стать пустым уже после вставки элемента x. Поэтому, чтобы увеличить эффективность данной реализации, необходимо в сегмент, который освободился после операции удаления элемента, поместить специальную константу, которую назовем *deleted* (удаленный). Важно различать константы *deleted* и *empty* — последняя находится в сегментах, которые никогда не содержали элементов. При таком подходе (даже при возможности удаления элементов) выполнение оператора MEMBER не требует просмотра всей хеш-таблицы. Кроме того, при вставке элементов сегменты, помеченные константой *deleted,* можно трактовать как свободные, таким образом, пространство, освобожденное после удаления элементов, можно рано или поздно использовать повторно. Но если невозможно непосредственно сразу после удаления элементов пометить освободившееся сегменты, то следует предпочесть закрытому хешированию схему открытого хеширования.

Пример 4.4. Предположим, что надо определить, есть ли элемент e в множестве, представленном на рис. 4.4. Если $h(e)$ — 4, то надо проверить еще сегменты 4, 5 и затем сегмент 6. Сегмент 6 пустой, в предыдущих просмотренных сегментах элемент e не найден, следовательно, этого элемента в данном множестве нет.

Допустим, мы удалили элемент c и поместили константу *deleted* в сегмент 4. В этой ситуации для поиска элемента d мы начнем с просмотра сегмент $h(d) = 3$, затем просмотрим сегменты 4 и 5 (где и найдем элемент d), при этом мы не останавливаемся на сегменте 4 — хотя он и пустой, но не помечен как *empty*. □

В листинге 4.9 представлены объявления типов данных и процедуры операторов для АТД DICTIONARY с элементами типа nametype и реализацией, использующей закрытую хеш-таблицу. Здесь используется хеш-функция Л из листинга 4.7, для разрешения коллизий применяется методика линейного хеширования. Константа *empty* определена как строка из десяти пробелов, а константа *deleted* — как строка из десяти символов "*", в предположении, что эти строки не совпадают ни с одним элементом словаря. (В базе данных примера 4.2 эти строки, очевидно, не будут совпадать с именами законодателей.) Процедура INSERT(x, Л) использует функцию *locate* (местонахождение) для определения, присутствует ли элемент x в словаре A или нет, а также специальную функцию *locate*1 для определения местонахождения элемента x. Последнюю функцию также можно использовать для поиска констант *deleted* и *empty*.

Листинг 4.9. Реализация словаря посредством закрытого хеширования

```
const
     empty = '          '; { 10 пробелов }
     deleted = '**********'; { 10 символов * }
type
     DICTIONARY = array[0..B-1] of nametype;

procerude MAKENULL ( var A: DICTIONARY );
     var
          i: integer;
     begin
          for i:= 0 to B - 1 do
               A[i]:= empty
     end,- { MAKENULL }

function locate ( x: nametype; A: DICTIONARY ): integer;
     { Функция просматривает A начиная от сегмента h(x) до тех
          пор, пока не будет найден элемент x или не встретится
```

```
        пустой сегмент или пока не будет достигнут конец таблицы
         (в последних случаях принимается, что таблица не содержит
        элемент x). Функция возвращает позицию, в которой
        остановился поиск. }
    var
        initial, i: integer;
    begin
        initial:= h(x);
        i:= 0;
        while (i < B) and (A[(initial + i) mod B] <> x) and
             (A[(initial + i) mod B] o empty) do
            i:= i + 1;
        return((initial + i) mod B)
    end; { locate }

function locatel ( x: nametype; A: DICTIONARY ): integer;
    { То же самое, что и функция locate, но останавливается и при
        достижении значения deleted }

function MEMBER ( x: nametype; var A: DICTIONARY ): boolean;
    begin
        if A[locate(x)] = x then
            return(true)
        else
            return(false)
    end; { MEMBER }

procedure INSERT ( x: nametype; var A: DICTIONARY );
    var
        backet: integer;
    begin
        if A[locate(x)] = x then return; { x уже есть в A }
        bucket:= locatel(x);
        if (A[bucket] = empty) or (A[bucket] = deleted) then
            A[bucket]:= x
        else
            error('Операция INSERT невозможна: таблица полна')
    end; { INSERT }

procedure DELETE ( x: nametype; var A: DICTIONARY );
    var
        bucket: integer;
    begin
        bucket:= locate(x);
        if A[locate(x)] = x then
            A[bucket]:= deleted
    end; { DELETE }
```

4.8. Оценка эффективности хеш-функций

Напомним, что хеширование — эффективный способ представления словарей и некоторых других абстрактных типов данных, основанных на множествах. В этом разделе мы оценим среднее время выполнения операторов словарей для случая открытого хеширования. Если есть B сегментов и N элементов, хранящихся в хеш-

таблице, то каждый сегмент в среднем будет иметь N/B элементов и мы ожидаем, что операторы INSERT, DELETE и MEMBER будут выполняться в среднем за время $O(1 + N/B)$. Здесь константа 1 соответствует поиску сегмента, а N/B— поиску элемента в сегменте. Если B примерно равно N, то время выполнения операторов становится константой, независящей от N.

Предположим, что есть программа, написанная на языке программирования, подобном Pascal, и мы хотим все имеющиеся в этой программе идентификаторы занести в хеш-таблицу. После обнаружения объявления нового идентификатора он вставляется в хеш-таблицу, конечно же, после проверки, что его еще нет в хеш-таблице. На этапе проверки естественно предположить, что идентификатор с равной вероятностью может быть в любом сегменте. Таким образом, на процесс заполнения хеш-таблицы с N элементами потребуется время порядка $O(N(1 + N/B))$. Если положить, что B равно N, то получим время $O(N)$.

На следующем этапе анализа программы просматриваются идентификаторы в теле программы. После нахождения идентификатора в теле программы, чтобы получить информацию о нем, его же необходимо найти в хеш-таблице. Какое время потребуется для нахождения идентификатора в хеш-таблице? Если время поиска для всех элементов примерно одинаково, то оно соответствует среднему времени вставки элемента в хеш-таблицу. Чтобы увидеть это, достаточно заметить, что время поиска любого элемента равно времени вставки элемента в конец списка соответствующего сегмента. Таким образом, время поиска элемента в хеш-таблице составляет $O(1 + N/B)$.

В приведенном выше анализе предполагалось, что хеш-функция распределяет элементы по сегментам равномерно. Но существуют ли такие функции? Рассмотрим функцию, код которой приведен в листинге 4.7, как типичную хеш-функцию. (Напомним, что эта функция преобразует символы в целочисленный код, суммирует коды всех символов и в качестве результата берет остаток от деления этой суммы на число B.) Следующий пример оценивает работу этой функции.

Пример 4.5. Предположим, что функция из листинга 4.7 применяется для занесения в таблицу со 100 сегментами 100 символьных строк АО, A1, ..., А99. Принимая во внимание, что $ord(0)$, $ord(1)$, ..., $ord(9)$ образуют арифметическую прогрессию (это справедливо для всех таблиц кодировок, где цифры 0, ..., 9 стоят подряд, например для кодировки ASCII), легко проверить, что эти элементы займут не более 29 сегментов из ста[1], наибольший сегмент (сегмент с номером 2) будет содержать элементы А18, А27, А36, ..., А90, т.е. девять элементов из ста[2]. Используя тот факт, что для вставки i-го элемента требуется $i + 1$ шагов, легко подсчитать, что в данном случае среднее число шагов, необходимое для вставки всех 100 элементов, равно 395. Для сравнения заметим, что оценка $N(1 + N/B)$ предполагает только 200 шагов. □

В приведенном примере хеш-функция распределяет элементы исходного множества по множеству сегментов не равномерно. Но возможны "более равномерные" хеш-функции. Для построения такой функции можно воспользоваться хорошо известным методом возведения числа в квадрат и извлечения из полученного квадрата нескольких средних цифр. Например, если есть число n, состоящее из 5 цифр, то после возведения его в квадрат получим число, состоящее из 9 или 10 цифр. "Средние цифры" — это цифры, стоящие, допустим, на позициях от 4 до 7 (отсчитывая справа). Их значения, естественно, зависят от числа n. Если $B = 100$, то для формирования номера сегмента достаточно взять две средние цифры, стоящие, например, на позициях 5 и 6 в квадрате числа.

---

[1] Отметим, что строки А2 и А20 не обязательно должны находиться в одном сегменте, но А23 и А41, например, будут располагаться в одном сегменте.

[2] Обратите внимание, что здесь "А" — буква английского алфавита и что приведенное распределение элементов по сегментам справедливо только для записанных выше символьных строк. Для другой буквы (или других символьных строк) получим другое распределение элементов по сегментам, но также, скорее всего, далекое от равномерного. — *Прим. ред.*

Этот метод можно обобщить на случай, когда В не является степенью числа 10. Предположим, что элементы исходного множества являются целыми числами из интервала О, 1, ..., K. Введем такое целое число С, что BC^2 примерно равно K^2, тогда функция

$$h(n) = [n^2/C] \bmod B,$$

где $[x]$ обозначает целую часть числа x, эффективно извлекает из середины числа n^2 цифры, составляющие число, не превышающее B.

Пример 4.6. Если $K = 1000$ и $B = 8$, то можно выбрать С = 354. Тогда

$$A(456) = \left[\frac{207936}{354}\right] \bmod 8 = 587 \bmod 8 = 3. \qquad \square$$

Для применения к символьной строке описанной хеш-функции надо сначала в строке сгруппировать символы справа налево в блоки с фиксированным количеством символов, например по 4 символа, добавляя при необходимости слева пробелы. Каждый блок трактуется как простое целое число, из которого путем конкатенации (сцепления) формируется двоичный код символов, составляющих блок. Например, основная таблица ASCII кодировки символов использует 7-битовый код, поэтому символы можно рассматривать как "цифры" по основанию 2^7 или 128[1]. Таким образом, символьную строку $abcd$ можно считать целым числом $(128)^3a + (128)^2b + (128)c + d$. После преобразования всех блоков в целые числа они суммируются[2], а затем выполняется вышеописанный процесс возведения в квадрат и извлечения средних цифр.

Анализ закрытого хеширования

В случае применения схемы закрытого хеширования скорость выполнения вставки и других операций зависит не только от равномерности распределения элементов по сегментам хеш-функцией, но и от выбранной методики повторного хеширования для разрешения коллизий, связанных с попытками вставки элементов в уже заполненные сегменты. Например, методика линейного хеширования для разрешения коллизий — не самый лучший выбор. Не приводя в данной книге полного решения этой проблемы, мы ограничимся следующим анализом.

Как только несколько последовательных сегментов будут заполнены (образуя группу), любой новый элемент при попытке вставки в эти сегменты будет вставлен в конец этой группы, увеличивая тем самым длину группы последовательно заполненных сегментов. Другими словами, для поиска пустого сегмента в случае непрерывного расположения заполненных сегментов необходимо просмотреть больше сегментов, чем при случайном распределении заполненных сегментов. Отсюда также следует очевидный вывод, что при непрерывном расположении заполненных сегментов увеличивается время выполнения вставки нового элемента и других операторов.

Определим, сколько необходимо сделать *проб* (проверок) на заполненность сегментов при вставке нового элемента, предполагая, что в хеш-таблице, состоящей из B сегментов, уже находится N элементов и все комбинации расположения N элементов в B сегментах равновероятны. Это общепринятые предположения, хотя не доказано, что схемы, отличные от схем закрытого хеширования, могут дать в среднем лучшее время выполнения операторов словарей. Далее мы получим формулы, оценивающие "стоимость" (количество проверок на заполненность сегментов) вставки нового элемента, если сегменты выбираются случайным образом. Наконец, мы рассмотрим не-

---

[1] Конечно, если вы работаете не только с латиницей, но и с кириллицей, то необходимо использовать полную таблицу ASCII. В этом случае основанием для "цифр"-символов будет не 2^7, а 2^8, но суть метода от этого не меняется. — *Прим. ред.*

[2] Если исходные символьные строки очень длинные, то суммирование можно выполнять по модулю некоторой константы с. Эту константу можно взять большей любого числа, получающегося из простого блока символов.

которые методики повторного хеширования, дающие "случайное" (равномерное) распределение элементов по сегментам.

Вероятность коллизии равна N/B. Предполагая осуществление коллизии, на первом этапе повторного хеширования "работаем" с B - 1 сегментом, где находится N - 1 элемент. Тогда вероятность возникновения двух подряд коллизий равна $N(N-1)/(B(B-1))$. Аналогично, вероятность по крайней мере i коллизий равна

$$\frac{N(N-1)\ldots(N-i+1)}{B(B-1)\ldots(B-i+1)}, \qquad (4.3)$$

Если значения B к N большие, то эта вероятность примерно равна $(N/B)^i$.Среднее Число проб равно единице плюс сумма по всем i > 1 вероятностей событий, что, по крайней мере, осуществится i коллизий, т.е. среднее число проб примерно равно (не превышает) $1+\sum_{i=1}^{\infty}(N/B)^i = \frac{B}{B-N}$.[1] Можно показать, что точное значение этой суммы при подстановке в нее выражения (4.3) вместо $(N/B)^i$ равно $\frac{B+1}{B+1-N}$, так что наша приближенная формула достаточно точна, за исключением случая, когда N близко к B.

Отметим, что величина $\frac{B+1}{B+1-N}$ растет очень медленно при возрастании N от 0 до B - 1, т.е. до максимального значения N, когда еще возможна вставка нового элемента. Например, если N равняется половине B, то в среднем требуются две пробы для вставки нового элемента. Среднее число проб на один сегмент при заполнении M сегментов равно $\frac{1}{M}\sum_{N=0}^{M-1}\frac{B+1}{B+1-N}$. Последнюю сумму можно аппроксимировать интегралом $\frac{1}{M}\int_0^{M-1}\frac{B}{B-x}dx$, который равен $\frac{B}{M}\ln\left(\frac{B}{B-M+1}\right)$. Таким образом, для полного заполнения таблицы ($M = B$) требуется в среднем $\ln B$ проб на сегмент, или всего $B\ln B$ проб. Но для заполнения таблицы на 90% ($M = 0.9B$) требуется всего $B((10/9)\ln 10)$, или примерно $2.56B$ проб.

При проверке на принадлежность исходному множеству элемента, которого заведомо нет в этом множестве, требуется в среднем такое же число проб, как и при вставке нового элемента при данном заполнении таблицы. Но проверка на принадлежность элемента, который принадлежит множеству, требует в среднем столько же проб, сколько необходимо для вставки всех элементов, сделанных до настоящего времени. Удаление требует в среднем примерно столько же проб, сколько и проверка элемента на принадлежность множеству. Но в отличие от схемы открытого хеширования, удаление элементов из закрытой хеш-таблицы не ускоряет процесс вставки нового элемента или проверки принадлежности элемента множеству. Необходимо особо подчеркнуть, что средние числа проб, необходимые для выполнения операторов словарей, являются константами, зависящими от доли заполнения хеш-таблицы. На рис. 4.5 показаны графики средних чисел проб, необходимые для выполнения операторов, в зависимости от доли заполнения хеш-таблицы.

---

[1] Здесь использована следующая формула вычисления среднего (математического ожидания). Пусть x — положительная целочисленная случайная величина, принимающая положительное целое значение i с вероятностью p_i. Тогда математическое ожидание случайной величины x можно вычислить по формуле $M[x] = \sum_{i=1}^{\infty} ip_i = \sum_{i=1}^{\infty} p_i + \sum_{i=2}^{\infty}\sum_{k=i}^{\infty} A = 1+\sum_{i=2}^{\infty} P\{x \geq i\}$. Отсюда легко получается приведенная в тексте оценка. — *Прим. ред.*

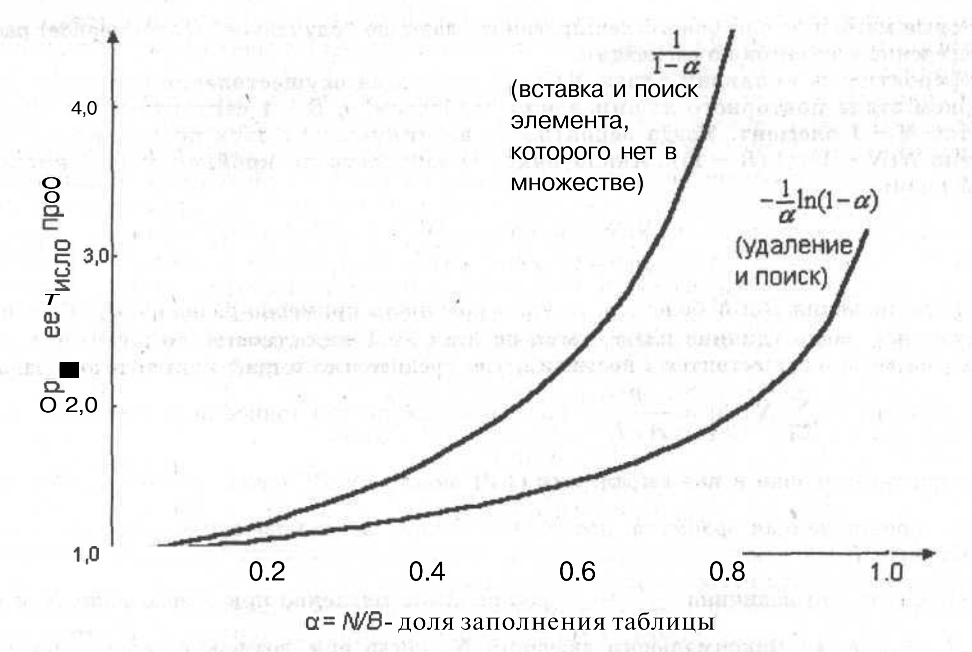


Рис. *4.5. Средние числа проб, необходимые для выполнения операторов*

„Случайные" **методики разрешения коллизий**

Мы ранее видели, что методика линейного повторного хеширования приводит к группированию заполненных сегментов в большие непрерывные блоки. Можно предложить хеш-функции с более "случайным" поведением, например, ввести целочисленную константу c > 1 и определить $h_i(x) = (h(x) + ci) \bmod B$. В этом случае для B = 8, $c = 3$ и $h(x) = 4$ получим "пробные" сегменты в следующем порядке: 4, 7, 2, 5, О, 3, 6 и 1. Конечно, если *B* **и** *c* имеют общие делители (отличные от единицы), то эта методика не позволит получить все номера сегментов, например при B = 8 и c = 2. Но более существенно, что даже если *B* **и** *c взаимно простые числа* (т.е. не имеют общих делителей), то все равно существует проблема "группирования", как и при линейном хешировании, хотя здесь разделяются блоки **заполненных** сегментов, соответствующие различным константам c. Этот феномен увеличивает время выполнения операторов словарей (как и при линейном хешировании), поскольку попытка вставить новый элемент в заполненный сегмент приводит к просмотру цепочки заполненных сегментов, различных для различных c, и длина этих цепочек при каждой вставке увеличивается на единицу.

Фактически любая методика повторного хеширования, где очередная проба зависит только от предыдущей (например, как зависимость от числа предыдущих "неудачных" проб, от исходного значения $h(x)$ или от самого элемента *x),* обнаруживает группирующие свойства линейного хеширования. Возможна простейшая методика, для которой проблема "группирования" не существует: для этого достаточно положить $h_i(x) = (h(x) + d_i) \bmod B$, где $d_1, d_2, ..., d_{B-1}$ — "случайные" перестановки чисел 1, 2, ..., B - 1. Конечно, такой набор чисел $d_1, d_2, ..., d_{B-1}$ должен использоваться при реализации всех операторов, выполняемых над словарями, а "случайное" перемешивание целых чисел должно быть сделано (выбрано) еще при разработке алгоритма хеширования.

Генерация "хороших случайных" чисел — достаточно сложная задача, но, к счастью, существуют сравнительно простые методы получения последовательности "случайных" чисел путем "перемешивания" целых чисел из заданного интервала.

При наличии такого генератора случайных чисел можно воспроизводить требуемую последовательность d_1, d_2, ..., d_{B-1} при каждом выполнении операторов, работающих с хеш-таблицей.

Одним из эффективных методов "перемешивания" целых чисел является метод "последовательных сдвигов регистра". Пусть B является степенью числа 2 и k — константа из интервала от 1 до B - 1. Начнем с некоторого числа d_1, взятого из интервала от 1 до B - *1.* Далее генерируется последовательность чисел d_i путем удвоения предыдущего значения до тех пор, пока последнее значение не превысит B. Тогда для получения следующего числа d_i из последнего значения отнимается число B и результат *суммируется побитово по модулю 2* с константой k. Сумма по модулю 2 чисел x и y (записывается как $x \oplus y$) определяется следующим образом: числа x и y записываются в двоичном виде с возможным приписыванием ведущих нулей так, чтобы числа имели одинаковую длину. Результат формируется по правилу логической операции "исключающего ИЛИ", т.е. 1 в какой-либо позиции результата будет стоять тогда и только тогда, когда только в одной аналогичной позиции слагаемых стоит 1, но не в обеих.

Пример 4.7. 25 ⊕ 13 вычисляется следующим образом:

25 = 11001
13 = 01101
25 ® 13 = 10100

Отметим, что такое "суммирование" возможно с помощью обычного двоичного суммирования, если игнорировать перенос разрядов. D

Не для каждого значения k можно получить все "перемешанные" значения из 1, 2, ..., B - *1,* иногда некоторые сгенерированные числа повторяются. Поэтому для каждого значения B необходимо подобрать свое значение k, которое будет "работать".

Пример 4.8. Пусть $B = 8$. Если мы положим $k = 3$, то сгенерируем все числа от О до 7. Например, если начнем с числа $d_1 = 5$, то для нахождения следующего числа d_2 сначала удваиваем число d_1, получаем число 10, которое больше 8. Поэтому из 10 вычитаем 8, получаем 2 и вычисляем d_2 = 2 Ф 3 = 1. Отметим, что для любого x сумму x Ф 3 можно вычислить путем инвертирования (преобразования в обратный код) последних двух двоичных разрядов числа x.

Повторяя описанную последовательность действий, получим числа d_1, d_2, ..., d_7 в виде 3-битовых двоичных чисел. Все этапы их вычисления показаны в табл. 4.2. Отметим, что умножение двоичного числа на 2 равнозначно сдвигу влево этого числа на один разряд — это оправдывает название "метод последовательного сдвига регистра".

Таблица 4.2. Вычисления по методу последовательного сдвига регистра

	d_1 = 101 = 5
сдвиг	1010
удаление ведущей 1 (вычитание 8)	010
Ф 3	d_2 = 001 = 1
сдвиг	da = 010 = 2
сдвиг	d_4 = 100 = 4
сдвиг	1000
удаление ведущей 1	000
Ф 3	di = 011 = 3
сдвиг	d_6 = 110 = 6
сдвиг	1100
удаление ведущей 1	100
⊕ 3	d_7 = 111 = 7

Читатель может проверить, что полный "перемешанный" набор чисел 1, 2, ..., 7 можно получить и для $k = 5$, но ни при каких других значениях ft. D

Реструктуризация хеш-таблиц

При использовании открытых хеш-таблиц среднее время выполнения операторов возрастает с ростом параметра N/B и особенно быстро растет при превышении числа элементов над числом сегментов. Подобным образом среднее время выполнения операторов также возрастает с увеличением параметра N/B и для закрытых **хеш-таблиц,** что видно из рис. 4.5 (но превышения N над B здесь невозможно).

Чтобы сохранить постоянное время выполнения операторов, которое теоретически возможно при использовании хеш-таблиц, мы предлагаем при достижении N достаточно больших значений, например при $N > 0.9B$ для закрытых хеш-таблиц и $N > 2B$ — для открытых хеш-таблиц, просто создавать новую хеш-таблицу с удвоенным числом сегментов. Перезапись текущих элементов множества в новую хеш-таблицу в среднем займет меньше времени, чем их ранее выполненная вставка в старую хеш-таблицу меньшего размера. Кроме того, затраченное время на перезапись компенсируется более быстрым выполнением операторов словарей.

4.9. Реализация АТД для отображений

Вернемся к АТД MAPPING (Отображение), введенным в главе 2, где мы определили отображение как функцию, ставящую в соответствие элементам области определения соответствующие элементы из области значений. Для этого АТД мы определили такие операторы.

1. **MAKENULL(*A*).** Инициализирует отображение *A*, где ни одному элементу области определения не соответствует ни один элемент области значений.
2. **ASSIGN(*A, d, r*).** Задает для *A(d)* значение *r*.
3. **COMPUTE(*A, d,* r).** Возвращает значение true и устанавливает значение *r* для *A*(*d*), если *A*(*d*) **определено,** в противном случае возвращает значение false.

Отображения можно эффективно реализовать с помощью хеш-таблиц. Операторы ASSIGN и COMPUTE реализуются точно так же, как операторы INSERT и MEMBER для словарей. Сначала рассмотрим применение открытых хеш-таблиц. Предполагаем, что хеш-функция $h(x)$ распределяет элементы области определения по сегментам хеш-таблицы. Так же, как и для словарей, в данном случае сегменты содержат связанные списки, в ячейках которых находятся поля для элементов области определения и для соответствующих им элементов области значений. Для реализации такого подхода надо в листинге 4.8 заменить определение типа ячеек следующим объявлением:

```
type
    celltype = record
        domainelement: domaintype;
        range: rangetype;
        next: ↑celltype
    end
```

Здесь domaintype и rangetype — типы данных для элементов области определения и области значений соответственно. Объявление АТД MAPPING следующее:

```
type
    MAPPING = array[0..B-1] of ↑celltype
```

Последний массив — это массив сегментов для хеш-таблицы. Код процедуры ASSIGN приведен в листинге 4.10. Написание процедур для операторов MAKENULL и COMPUTE оставляем для упражнения.

Листинг 4.10. Процедура ASSIGN для открытой хеш-таблицы

```
procedure ASSIGN ( var A: MAPPING; d: domaintype; r: rangetype );
    var
        bucket: integer;
        current: ↑celltype;
    begin
        bucket:= h(d);
        current:= A[bucket];
        while current <> nil do
            if current↑.domainelement = d then begin
                current↑.range:= r;
                    { замена старого значения для d }
                return
            end
            else
                current:= current↑.next; { d не найден в списке }
        current:= A[bucket];
            {использование current для запоминания первой ячейки}
        new(A[bucket]);
        A[bucket]↑.domainelement:= d;
        A[bucket]↑.range:= r;
        A[bucket]↑.next:= current;
    end; { ASSIGN }
```

Подобным образом для реализации отображений можно использовать закрытое хеширование. Ячейки хеш-таблицы будут содержать поля для элементов областей определения и значений, а АТД MAPPING можно определить как массив таких ячеек. Как и в случае открытого хеширования, хеш-функция применяется к элементам области определения, а не к элементам области значений. Мы оставляем читателю в качестве упражнения реализацию соответствующих операторов для закрытых хеш-таблиц.

4.10. Очереди с приоритетами

Очередь с приоритетами — это АТД, основанный на модели множеств с операторами INSERT и DELETEMIN, а также с оператором MAKENULL для инициализации структуры данных. Перед определением нового оператора DELETEMIN сначала объясним, что такое "очередь с приоритетами". Этот термин подразумевает, что на множестве элементов задана функция приоритета (priority), т.е. для каждого элемента a множества можно вычислить функцию $p(a)$, *приоритет элемента a*, которая обычно принимает значения из множества действительных чисел, или, в более общем случае, из некоторого линейно упорядоченного множества. Оператор INSERT для очередей с приоритетами понимается в обычном смысле, тогда как DELETEMIN является функцией, которая возвращает элемент с наименьшим приоритетом и в качестве побочного эффекта удаляет его из множества. Таким образом, оправдывая свое название, DELETEMIN является комбинацией операторов DELETE и MIN, которые были описаны выше в этой главе.

Пример 4.9. Название "очередь с приоритетами" происходит от того вида упорядочивания (сортировки), которому подвергаются данные этого АТД. Слово "очередь" предполагает, что люди (или входные элементы) ожидают некоторого обслуживания,

а слова "с **приоритетом**" обозначают, что обслуживание будет производиться не по принципу "первый пришел — первым получил обслуживание", как происходит с АТД QUEUE (Очередь), а на основе приоритетов всех персон, стоящих в очереди. Например, в приемном отделении больницы сначала принимают пациентов с потенциально фатальными диагнозами, независимо от того, как долго они или другие пациенты находились в очереди.

Более приемлемым для данной книги будет пример очереди с приоритетами, которая возникает среди множества процессов, ожидающих обслуживания совместно используемыми ресурсами компьютерной системы. Обычно системные разработчики стараются сделать так, чтобы короткие процессы выполнялись незамедлительно (на практике "незамедлительно" может означать одну-две секунды), т.е. такие процессы получают более высокие приоритеты, чем процессы, которые требуют (или уже израсходовали) значительного количества системного времени. Процессы, которые требуют нескольких секунд машинного времени, не выполняются сразу — рациональная стратегия разделения ресурсов откладывает их до тех пор, пока не будут выполнены короткие процессы. Однако нельзя переусердствовать в применении этой стратегии, иначе процессы, требующие значительно больше времени, чем "средние" процессы, вообще никогда не смогут получить кванта машинного времени и будут находиться в режиме ожидания вечно.

Один возможный путь удовлетворить короткие процессы и не заблокировать большие состоит в задании процессу P приоритета, который вычисляется по формуле $100t_{исп}(P) - t_{нач}(P)$. Здесь параметр $t_{исп}$ равен **времени**, израсходованному процессом P ранее, а $t_{нач}$ — время, прошедшее от начала инициализации процесса, отсчитываемое от некоего "нулевого времени". Отметим, что в общем случае приоритеты будут большими отрицательными целыми числами, если, конечно, $t_{нач}$ не отсчитывается от "нулевого времени" в будущем. Число 100 в приведенной формуле является "магическим числом" (т.е. не поддается четкому логическому обоснованию, а пришедшему из практики) и должно быть несколько больше числа ожидаемых активных процессов. Читатель может легко увидеть, что если всегда сначала выполняется процесс с наименьшим приоритетным числом и если в очереди немного коротких процессов, то в течение некоторого (достаточно продолжительного) времени процессу, который не закончился за один квант машинного времени, будет выделено не менее 1% машинного времени. Если этот процент надо увеличить или уменьшить, следует заменить константу 100 в формуле вычисления приоритета.

Представим процесс в виде записи, содержащей поле id идентификатора процесса и поле *priority* со значением приоритета, т.е. тип процесса processtype определим следующим образом:

```
type
    processtype = record
        id: integer;
        priority: integer
    end;
```

Значение приоритета мы определили как целое число. Функцию определения приоритета (но не его вычисления) можно определить так:

```
function p ( a: processtype ): integer;
    begin
        return(a.priority)
    end;
```

Для того чтобы для выбранных процессов зарезервировать определенное количество квантов машинного времени, компьютерная система поддерживает очередь с приоритетом WAITING (Ожидание), состоящую из элементов типа processtype и использующую две процедуры: *initial* (инициализация) и *select* (выбирать). Очередь WAITING управляется с помощью операторов INSERT и DELETEMIN. При инициа-

лизации нового процесса вызывается процедура *initial*, которая указывает записи, соответствующей новому процессу, место в очереди WAITING. Процедура *select* вызывается тогда, когда система хочет "одарить" квантом машинного времени какой-либо процесс. Запись для выбранного процесса удаляется из очереди WAITING, но сохраняется посредством *select* для повторного ввода в очередь (если процесс не закончился за выделенное время) с новым приоритетом — приоритет увеличивается на 100 при пересчете времени $t_{исп}$ (когда $t_{исп}$ увеличивается на единицу).

Можно использовать функцию *currenttime* (которая возвращает текущее машинное время) для вычисления временных интервалов, отводимых системой процессам; обычно эти интервалы измеряются в микросекундах. Будем также использовать процедуру *execute*(*P*) для вызова процесса с идентификатором *P* на исполнение в течение одного кванта времени. В листинге 4.11 приведены коды процедур *initial* и *select*. □

Листинг 4.11. Выделение процессам машинного времени

```
procedure initial ( P: integer );
{initial указывает процессу с идентификатором P место в очереди}
    var
        process: processtype;
    begin
        process.id:= P;
        process.priority:= - currenttime;
        INSERT(process, WAITING)
    end; { initial }

procedure select;
{select выделяет квант времени процессу с наивысшим приоритетом}
    var
        begintime, endtime: integer;
        process: processtype;
    begin
        process:=↑DELETEMIN(WAITING);
            {DELETEMIN возвращает указатель на удаленный элемент}
        begintime:= currenttime;
        execute (process.id);
        endtime:= currenttime;
        process.priority:=process.priority+100*(endtime-begintime);
            { пересчет приоритета }
        INSERT(process,WAITING)
            { занесение процесса в очередь с новым приоритетом }
    end; { select }
```

4.11. Реализация очередей с приоритетами

За исключением хеш-таблиц, те реализации множеств, которые мы рассмотрели ранее, можно применить к очередям с приоритетами. Хеш-таблицы следует исключить, поскольку они не имеют подходящего механизма нахождения минимального элемента. Попросту говоря, применение хеширования привносит дополнительные сложности, которых лишены, например, связанные списки.

При использовании связанных списков можно выбрать вид упорядочивания элементов списка или оставить его несортированным. Если список отсортирован, то нахождение минимального элемента просто — это первый элемент списка. Но вместе с тем вставка нового элемента в отсортированный список требует просмотра в среднем

половины элементов списка. Если оставить список неупорядоченным, упрощается вставка нового элемента и затрудняется поиск минимального элемента.

Пример 4.10. В листинге 4.12 показана реализация функции DELETEMIN для несортированного списка элементов, имеющих тип processtype, описанный в примере 4.9. В этом листинге также приведены объявления для типа ячеек списка и для АТД PRIORITYQUEUE (Очередь с приоритетами). Реализация операторов INSERT и MAKENULL (как для отсортированных, так и для несортированных списков) не вызывает затруднений, и мы оставляем ее в качестве упражнения для читателей. □

Листинг 4.12. Реализация очереди с приоритетами посредством связанного списка

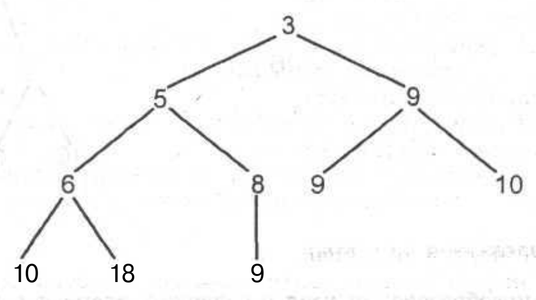
```
type
    celltype = record
        element: processtype;
        next: ↑celltype
    end;
    PRIORITYQUEUE = Tcelltype;
        { ячейка указывает на заголовок списка }

function DELETEMIN ( var A: PRIORITYQUEUE ): Tcelltype;
    var
        current: Tcelltype;
            { указывает на ячейку, которая будет
                проверена следующей }
        lowpriority: integer;
            { содержит ранее найденный наименьший приоритет }
        prewinner: Tcelltype;
            { указывает на ячейку, содержащую элемент
                с наименьшим приоритетом }
    begin
        if A↑.next = nil then
            error('Нельзя найти минимум в пустом списке')
        else begin
            lowpriority:= p(A↑.next↑.element);
                { функция p возвращает приоритет первого элемента.
                    Отметим, что Л указывает на ячейку заголовка,
                    которая не содержит элемента }
            prewinner:= A;
            current:= A↑.next;
            while current↑.next <> nil do begin
                { сравнение текущего наименьшего приоритета с
                    приоритетом следующего элемента }
                if p(current↑.next↑.element)<lowpriority then begin
                    prewinner:= current;
                    lowpriority:= p (current↑.next↑.element)
                end;
                сиггелt:= current↑.next
            end;
            DELETEMIN:= prewinner↑.next;
                { возвращает указатель на найденный элемент }
            prewinner↑.next:= prewinner↑.next↑.next
                { удаляет найденный элемент из списка }
        end
    end; { DELETEMIN }
```

Реализация очереди с приоритетами посредством частично упорядоченных деревьев

Какие бы мы ни выбрали списки, упорядоченные или нет, для представления очередей с приоритетами придется затратить время, пропорциональное n, для выполнения операторов INSERT и DELETEMIN на множестве размера n. Существует другая реализация очередей с приоритетами, в которой на выполнение этих операторов требуется порядка $O(\log n)$ шагов, что значительно меньше n для больших n (скажем, для $n > 100$). Основная идея такой реализации заключается в том, чтобы организовать элементы очереди в виде сбалансированного[1] (по возможности) двоичного дерева (рис. 4.6). На нижнем уровне, где некоторые листья могут отсутствовать, мы требуем, чтобы все отсутствующие листья в принципе могли располагаться только справа от присутствующих листьев нижнего уровня.

Более существенно, что дерево *частично упорядочено*. Это означает, что приоритет узла v не больше приоритета любого сына узла v, где приоритет узла — это значение приоритета элемента, хранящегося в данном узле. Из рис. 4.6 видно, что малые значения приоритетов не могут появиться на более высоком уровне, где есть большие значения приоритетов. Например, на третьем уровне располагаются приоритеты 6 и 8, которые меньше приоритета 9, расположенного на втором уровне. Но родитель узлов с приоритетами 6 и 8, расположенный на втором уровне, имеет (и должен иметь) по крайней мере не больший приоритет, чем его сыновья.



Рис, 4.6. Частично упорядоченное дерево

При выполнении функции DELETEMIN возвращается элемент с минимальным приоритетом, который, очевидно; должен быть корнем дерева. Но если мы просто удалим корень, то тем самым разрушим дерево. Чтобы не разрушить дерево и сохранить частичную упорядоченность значений приоритетов на дереве после удаления корня, мы делаем следующее: сначала находим на самом нижнем уровне самый правый узел и временно помещаем его в корень дерева. На рис. 4.7,а показаны изменения, сделанные на дереве рис. 4.6 после удаления корня. Затем этот элемент мы спускаем по ветвям дерева вниз (на более низкие уровни), по пути меняя его местами с сыновьями, имеющими меньший приоритет, до тех пор, пока этот элемент не станет листом или не встанет в позицию, где его сыновья будут иметь по крайней мере не меньший приоритет.

На рис. 4.7,а надо поменять местами корень и его сына, имеющего меньший приоритет, равный 5. Результат показан на рис. 4.7,б. Наш элемент надо спустить еще

---

[1] Сбалансированность в данном случае конструктивно можно определить так: листья возможны только на самом нижнем уровне или на предыдущем, но не на более высоких уровнях. Другими словами, максимально возможная сбалансированность двоичного дерева здесь понимается в том смысле, чтобы дерево было как можно "ближе" к полному двоичному дереву. Отметим также, что это понятие сбалансированности дерева не следует путать с α-сбалансированностью деревьев и подобными характеристиками, хотя они и находятся в определенном "родстве". — *Прим. ред.*

на более низкий уровень, так как его сыновья сейчас имеют приоритеты 6 и 8. Меняем его местами с сыном, имеющим наименьший приоритет 6. Полученное в результате такого обмена новое дерево показано на рис. 4.7,в. Это дерево уже является частично упорядоченным и его дальше не надо менять.

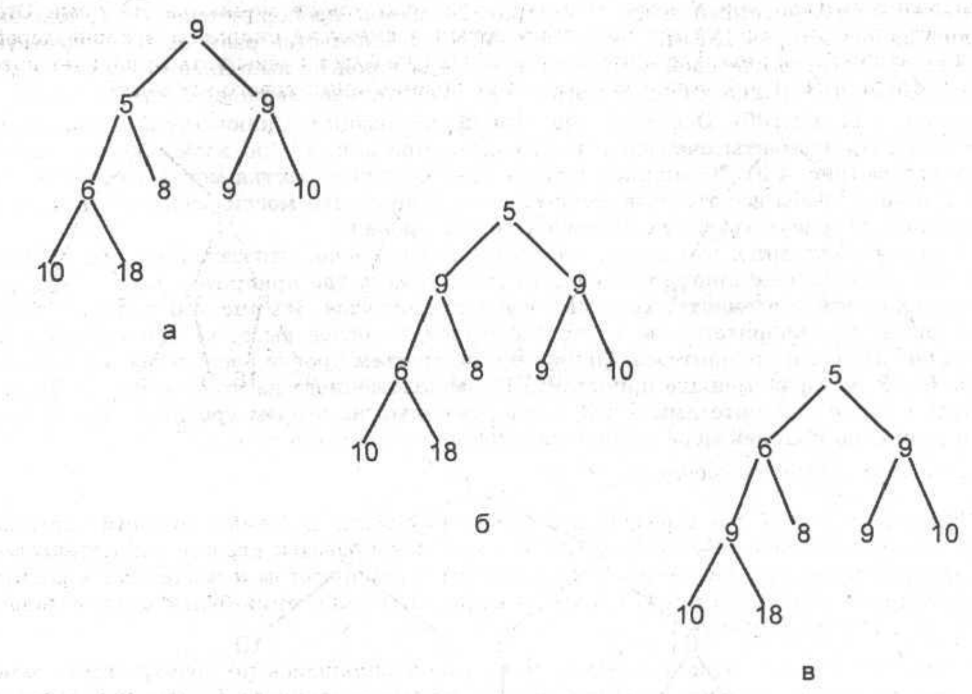


Рис. 4.7. Спуск элемента по дереву

Если при таком преобразовании узел v содержит элемент с приоритетом a и его сыновьями являются элементы с приоритетами b и c, из которых хотя бы один меньше а, то меняются местами элемент с приоритетом а и элемент с наименьшим приоритетом из & и с. В результате в узле v будет находиться элемент с приоритетом, не превышающим приоритеты своих сыновей. Чтобы доказать это, для определенности положим, что $b \leq c$. После обмена элементами узел v будет содержать элемент с приоритетом b, a его сыновья — элементы с приоритетами а и с. Мы предположили, что $b <$ с и а не меньше, по крайней мере, или b, или c. Отсюда следует $b < a$, что и требовалось доказать. Таким образом, описанный процесс спуска элемента по дереву приводит к частичному упорядочиванию двоичного дерева.

Теперь покажем, что оператор DELETEMIN, выполняемый над множеством из n элементов, требует времени порядка $O(\log n)$. Это следует из того факта, что в дереве нет пути, состоящего из больше чем $1 + \log n$ узлов[1], а также вследствие того, что процесс прохождения элементом каждого узла занимает постоянное фиксированное время. Так как для любой положительной константы c и при $n > 2$ величина $c(1 + \log n)$ не превышает $2c\log n$, то $c(1 + \log n)$ имеет порядок $O(\log n)$.

Теперь рассмотрим, как на частично упорядоченных деревьях работает оператор INSERT. Сначала поместим новый элемент в самую левую свободную позицию на самом нижнем уровне, если же этот уровень заполнен, то следует начать новый уро-

---

[1] Здесь подразумевается логарифм по основанию 2. Приведенную оценку можно доказать исходя из упражнения 3.18 главы 3 или непосредственно методом математической индукции по *n*. — *Прим. ред.*

вень. На рис. **4.8,а** показана вставка элемента с приоритетом 4 в дерево из рис. 4.6. Если новый элемент имеет меньший приоритет, чем у его родителя, то они меняются местами. Таким образом, новый элемент теперь находится в позиции, когда у его сыновей больший приоритет, чем у него. Но возможно, что у его нового родителя приоритет больше, чем у него. В этом случае они также меняются местами. Этот процесс продолжается до тех пор, пока новый элемент не окажется в корне дерева или не займет позицию, где приоритет родителя не будет превышать приоритет нового элемента. Рис. **4.8,б,** в показывают этапы перемещения нового элемента.

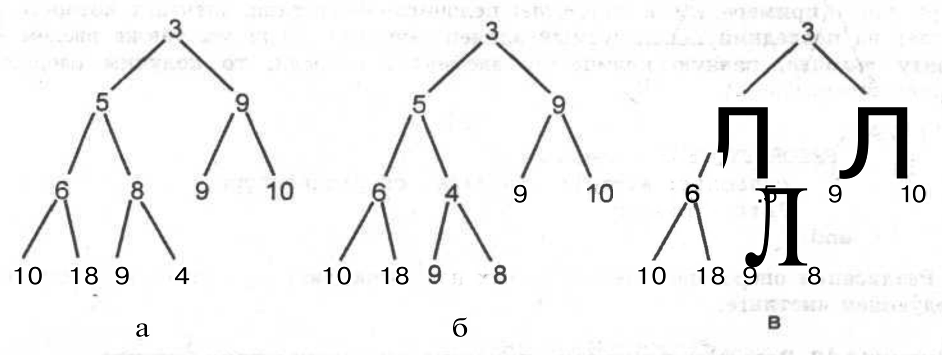


Рис. 4.8. Вставка нового элемента

Теперь докажем, что в результате описанных выше действий получим частично упорядоченное дерево. Поскольку мы не пытаемся провести строгое доказательство, то просто рассмотрим ситуации, когда элемент с приоритетом *а* может стать родителем элемента с приоритетом *Ь*. (Далее для простоты изложения будем отождествлять элемент с его приоритетом.)

1. Элемент *а* — это новый элемент, который, перемещаясь по дереву вверх, заменил родителя элемента *Ь*. Пусть старый родитель элемента *Ь* имел приоритет *с*, тогда *а* < *с*, иначе не произошла бы замена. Но *с* < *Ь*, поскольку исходное дерево частично упорядочено. Отсюда следует, что *а* < *Ь*. Пример такой ситуации показан на рис. **4.8,в,** где элемент 4 становится родителем элемента 6, заменяя его родителя с приоритетом 5.
2. Элемент *а* спустился вниз по дереву вследствие обмена с новым элементом. В этом случае в исходном дереве элемент а должен быть предком элемента *Ь*. Поэтому *а* < *Ъ*. На рис. **4.8,в** элемент **5,** ставший родителем элементов с приоритетами 8 и 9, ранее был их "дедушкой".
3. Элемент *Ь* — новый элемент, который, перемещаясь вверх по дереву, стал сыном элемента *а*. Если *а* > *Ь*, то на следующем шаге они поменяются местами, ликвидируя тем самым нарушение свойства упорядоченности **дерева**.

Время выполнения оператора вставки пропорционально пути, пройденному новым элементом. Так же, как и в случае оператора DELETEMIN, этот путь не может быть больше $1 + \log n$. Таким образом, время выполнения обоих этих операторов имеет порядок $O(\log n)$.

Реализация частично упорядоченных деревьев посредством массивов

Исходя из того, что рассматриваемые нами деревья являются двоичными, по возможности сбалансированными и на самом нижнем уровне все листья "сдвинуты" влево, можно применить для этих деревьев необычное представление, которое называется *куча*. В этом представлении используется массив, назовем его *А*, в котором первые n позиций соответствуют га узлам дерева. $A[1]$ содержит корень дерева. Левый сын узла $A[i]$, если он существует, находится в ячейке $A[2i]$, а правый сын, если он

также существует, — в ячейке $A[2i + 1]$. Обратное преобразование: если сын находится в ячейке $A[i]$, $i > 1$, то его родитель — в ячейке $A[i \text{ div } 2]$. Отсюда видно, что узлы дерева заполняют ячейки $A[1], A[2], ..., A[n]$ последовательно уровень за уровнем, начиная с корня, а внутри уровня — слева направо. Например, дерево, показанное на рис. 4.6, будет представлено в массиве следующей последовательностью своих узлов: 3, 5, 9, 6, 8, 9, 10, 10, 18, 9.

В данном случае мы можем объявить АТД PRIORITYQUEUE (Очередь с приоритетами) как записи с полем *contents* для массива элементов типа, например processtype, как в примере 4.9, и полем *last* целочисленного типа, значение которого указывает на последний используемый элемент массива. Если мы также введем константу *maxsize*, равную количеству элементов очереди, то получим следующее объявление типов:

```
type
    PRIORITYQUEUE = record
        contents: array[1..maxsize] of processtype;
        last: integer
    end;
```

Реализация операторов, выполняемых над очередью с приоритетами, показана в следующем листинге.

Листинг 4.13. Реализация очереди с приоритетами посредством массива

```
procedure MAKENULL ( var Л: PRIORITYQUEUE );
    begin
        A.last:= 0
    end; { MAKENULL }

procedure INSERT ( x: processtype; var A: PRIORITYQUEUE ) ,-
    var
        i: integer;
        temp: processtype;
    begin
        if A.last >= maxsize then
            error('Очередь заполнена')
        else begin
            A.last:= A.last + 1;
          • A,contents[A.last]:= x;
            i:= A.last; { i — индекс текущей позиции x }
            while (i > 1) and (p(A.contents[i]) <
                                p(A.contents[i div 2])) do
                begin { перемещение x вверх по дереву путем обмена
                местами с родителем, имеющим больший приоритет }
                temp:= A.contents[i];
                A.contents[i]:= A.contents[i div 2];
                A.contents[i div 2]:= temp;
                i:= i div 2
            end
        end
    end; { INSERT }

function DELETEMIN ( var A: PRIORITYQUEUE ): ↑processtype;
    var
        i, j: integer;
        temp: processtype;
```

```
    minimum: ↑processtype;
begin
    if A.last = 0 then
        error('Очередь пуста')
    else begin
        new(minimum);
        minimum T := A.contents[1];
        A. contents[1]:= A.contents[A.last];
        A.last:= A.last - 1;
        i:= 1;
        while i <= A.last div 2 do begin
            { перемещение старого последнего элемента
                вниз по дереву }
            if (p(A.contents[2*i]) < p(A.contents[2*i + 1]))
                or (2*i = A.last) then
                    j:= 2*i
            else
                j:= 2*i + 1;
            { j будет сыном i с наименьшим приоритетом или,
                если 2*i = A.last, будет просто сыном i }
            if p(A.contents[i]) > p(A.contents[j]) then begin
                { обмен старого последнего элемента с сыном,
                    имеющим наименьший приоритет }
                temp:= A.contents[i] ;
                A.contents[i]:= A.contents[j];
                A.contents[j]:= temp;
                i:= j;
            end
            else
                return(minimum)
                    { дальше перемещение элемента невозможно }
        end;
        return(minimum) { элемент дошел до листа }
    end
end; { DELETEMIN }
```

4.12. Некоторые структуры сложных множеств

В этом разделе мы рассмотрим применение двух сложных множеств для представления данных. В первом примере рассмотрим представление отношений "многие-ко-многим", которые встречаются в системах баз данных. Далее мы изучим, как с помощью двух структур данных можно представить какой-либо объект (в нашем случае — отображение) более эффективно, чем при представлении этого объекта одной структурой.

Отношения „многие-ко-многим" и структура мультисписков

Пример отношения "многие-ко-многим" между множеством студентов и множеством учебных курсов (учебных предметов) показан в табл. 4.3. Отношение "многие-ко-многим" называется так потому, что на каждый курс могут записаться много студентов (не один) и каждый студент может записаться на несколько курсов.

Время от времени может возникнуть необходимость вставить или удалить студентов с какого-либо курса, определить, какие студенты посещают тот или ной курс, или узнать, какие курсы посещает конкретный студент. Простейшей структурой

данных, которая удовлетворяет этим запросам, является двумерный массив, подобный табл. 4.3, где значение 1 (или true) соответствует значку "X", а значение О (или **false**) — пробелу.

Таблица 4.3. Пример отношения между множеством студентов и множеством учебных курсов

	CS101	CS202	CS303
Alan			X
Alex	X	X	
Alice			X
Amy	X		
Andy		X	X
Ann	X		X

Регистрация (студентов по учебным курсам)

Чтобы приписать студента к какому-либо курсу, надо использовать отображение (назовем его MS), которое можно реализовать как хеш-таблицу, для преобразования имени студента в индекс массива, а также еще одно отображение *MC,* переводящее название курса в другой индекс массива. Тогда запись студента *s* на курс *c* можно представить в виде простого присваивания элементу массива *Enrollment* (Регистрация) значения 1:

$$Enrollment[MS(s),MC(c)] := 1.$$

Открепление студента с курса выполняется путем задания значения О соответствующему элементу массива *Enrollment.* Для определения курсов, которые посещает студент с именем s, надо просмотреть строку *MS(s),* а для получения списка студентов, посещающих курс *c,* надо просмотреть столбец *MC(c).*

Почему для представления подобных данных надо использовать другую, более подходящую структуру данных? Представим большой университет, где учатся примерно 20 000 студентов и изучается не менее 1 000 учебных курсов, при этом каждый студент одновременно изучает в среднем только три учебных курса. Тогда при использовании массива, подобного табл. 4.3, этот массив будет состоять из 20 000 000 элементов, из которых только 60 000, или примерно 0.3%, будут иметь значение 1[1]. Такой массив называют *разреженным,* так как большинство его элементов имеют нулевые значения. Можно сберечь значительный объем памяти, если хранить не весь разреженный массив, а только его ненулевые элементы. Более того, в этом случае можно значительно сократить время просмотра этого массива. Например, вместо просмотра столбца из 20 000 элементов надо просмотреть в среднем всего 60 элементов. Просмотр элементов по строкам займет примерно такое же время.

Еще один подход к решению исходной задачи состоит в ее переформулировке: вместо представления данных, подобных табл. 4.3, в виде одного множества можно создать систему поддержки набора множеств и отношений между ними. В нашем случае очевидны два множества: имен студентов S и названий учебных курсов C. Каждый элемент множества S будет иметь тип **studenttype** (тип студента), который можно представить в виде записей следующего типа:

```
type
    studenttype = record
        id: integer;
        name: array[1..30] of char;
    end
```

[1] Если бы это была реальная база данных, то такой массив должен храниться на устройстве внешней памяти. Но такая структура данных слишком расточительна даже для внешней памяти.

Подобное определение можно сделать для элементов множества *C*. Для реализации отношений между элементами множеств необходимо третье множество *E*, каждый элемент которого должен соответствовать одной ячейке массива регистрации, где есть знак "X" (см. табл. 4.3). Элементы множества *E* должны быть записями фиксированного типа. Мы пока не можем сказать, какие поля должны содержать эти записи[1], но далее мы рассмотрим несколько возможных вариантов таких полей. Сейчас мы просто постулируем, что есть регистрационные записи для каждой ячейки массива, помеченной знаком "X", и эти записи каким-то образом отделены одна от другой.

Нам также нужны множества, соответствующие ответам на основные вопросы: какие учебные курсы посещает конкретный студент s (это множество обозначим C_s) *и* какие студенты записаны на данный курс *c* (это множество S_c). Реализации этих множеств вызывают затруднения, поскольку заранее нельзя сказать, сколько будет элементов в этих множествах, что, в свою очередь, принуждает усложнять записи, касающиеся студентов и курсов. Можно сделать множества C_s и S_c не множествами непосредственно записей, а множествами указателей на студенческие и курсовые записи. В этом случае экономится значительная часть пространства памяти и можно быстро получить ответы на сформулированные выше вопросы.

Пока пусть каждое множество C_s является множеством регистрационных записей, соответствующих конкретному студенту s и некоторому учебному курсу c. Если принять за регистрацию пару (s, *c)*, то множество C_s можно определить следующим образом:

$$C_s = \{(s, c) \mid \text{студент s записан на курс } c\}.$$

Аналогично определяется множество S_c:

$$S_c = \{(s, c) \mid \text{студент s записан на курс } c\}.$$

Отметим различие в определении этих множеств: в первом множестве *s* является константой, а во втором — *c*. Например, основываясь на табл. 4.3, C_{Alex} = {(Alex, CS101), (Alex, CS202)} и S_{CS101} = {(Alex, CS101), (Amy, CS101), (Ann, CS101)}.

Структуры мультисписков

В общем случае структура мультисписка — это совокупность ячеек, некоторые из которых имеют более одного указателя и поэтому одновременно могут принадлежать нескольким спискам. Для каждого типа ячеек важно различать поля указателей для разных списков, чтобы можно было проследить элементы одного списка, не вступая в противоречие с указателями другого списка.

С этой точки зрения можно задать поля указателя в каждой записи для студента и для курса, которые указывали бы на первые регистрационные записи множеств C_s и S_c соответственно. Каждая регистрационная запись должна иметь два поля указателя, одно из них, назовем его *cnext*, будет указывать на следующую регистрационную запись списка множества C_s, которому она принадлежит, а второе поле, *snext*, будет указывать на следующую регистрационную запись списка множества S_c, которому она также принадлежит.

Оказывается, что при таком задании полей указателей в регистрационных записях они могут не указывать непосредственно ни на студента, ни на курс, которым она (регистрационная запись) соответствует. Эта информация неявно содержится в списках, которым принадлежит регистрационная запись. Назовем студенческие и курсовые записи, возглавляющие такие списки, *собственниками* регистрационной записи. Таким образом, чтобы найти те курсы, которые посещает студент *s*, надо просмотреть все регистрационные записи из C_s и для каждой из них найти собственника среди курсовых записей. Для этого достаточно поместить в каждую регистрационную запись указатели на обоих собственников.

---

[1] На практике в записях, подобных регистрационным, удобно иметь поля типа метки или статуса, но наша исходная задача этого не требует.

Благодаря таким указателям можно получить ответы на интересующие нас вопросы за минимально возможное время. Но можно значительно сократить объем памяти, занимаемой такой **структурой**[1], если исключить описанные выше указатели на собственников из регистрационных записей, оставив их только в конечных записях списков каждого C_s и S_c. Таким образом, каждая запись о студенте или курсе становится частью кольца, включающего все регистрационные записи, для которых данная запись о студенте или курсе является собственником. Такие кольца можно видеть на рис. 4.9 для данных из табл. 4.3. Отметим, что. на этом рисунке регистрационные записи состоят из поля ***cnext*** (левое) и поля ***snext*** (правое).

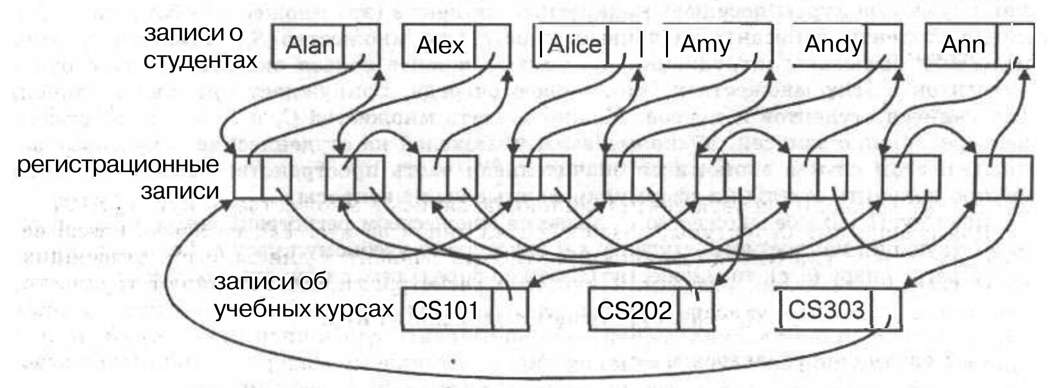


Рис. 4.9. Представление данных из табл. 4.3 посредством мультисписка

Пример 4.11. Для ответа на вопрос, какие студенты зарегистрированы на курсе CS101, сначала находим запись для этого учебного курса. Как найти эту запись, зависит от организации множества курсов. Например, это может быть хеш-таблица, содержащая все курсовые записи, тогда определить нужную запись можно посредством хеш-функции, примененной к "CS101".

Далее следуем за указателем из записи курса CS101 на первую регистрационную запись кольца для этого курса. На рис. 4.9 это вторая слева регистрационная запись. Теперь надо найти собственника-запись студента, которому принадлежит данная регистрационная запись. Для этого мы следуем за указателями *cnext* (первый указатель в регистрационной записи), пока не достигнем указателя на запись студента[2]. В данном случае мы остановимся на третьей регистрационной записи, которая указывает на запись студента Alex. Таким образом, мы узнали, что Alex посещает учебный курс CS101.

Теперь надо найти следующего студента, посещающего курс CS101. Для этого надо следовать за указателем *snext* (второй указатель в регистрационной записи), начиная со второй регистрационной записи. В данном случае он приведет к пятой регистрационной записи. В этой записи указатель *cnext* непосредственно указывает на собственника-запись студента, которая соответствует студенту Amy. Таким образом, студент Amy также посещает курс CS101. Следуя далее за указателем *snext* пятой регистрационной записи, мы перейдем к восьмой регистрационной записи. Кольцо указателей *cnext* приведет от этой записи к девятой регистрационной записи, собственником которой является запись, соответствующая студентке Ann, также посещающей курс CS101. Указатель *snext* восьмой регистрационной записи возвращает нас на запись курса CS101. Это означает, что множество S_c исчерпано. □

---

[1] Отметим, что, как правило, регистрационных записей значительно больше, чем записей о студентах и учебных курсах, поэтому, даже незначительно сокращая пространство, занимаемое одной регистрационной записью, мы существенно сокращаем общее требуемое пространство памяти.

[2] Чтобы отличить указатель на регистрационную запись от указателя на запись студента, надо как-то определять тип записей. Этот вопрос мы рассмотрим немного ниже.

В обобщенном виде операторы, реализующие действия, описанные в примере 4.11, можно записать следующим образом:

```
for для каждой записи регистрации из множества Sc курса CS101 do
    begin
    s: = студент-собственник регистрационной записи;
    print(s)
end
```

Здесь оператор присваивания можно детализировать так:

```
f: = e;
repeat
    f:= f↑.cnext
until
    f — указатель на запись студента;
s:= значению поля studentname в записи, на которую указывает f;
```

где *e* — указатель на первую регистрационную запись в множестве S_c курса CS101.

Для реализации структуры, подобной показанной на рис. 4.9, на языке Pascal необходим только тип record (запись) в различных вариантах для записей, относящихся к именам студентов, названиям курсов и регистрации. Но возникают трудности, связанные с тем, что указатели полей *cnext* и *snext* могут указывать записи разных типов. Эти трудности можно обойти, если определять тип записи циклически. В листинге 4.14 показаны такие объявления типов, а также процедура *printstudents* вывода на печать имен студентов, посещающих определенный учебный курс.

Листинг 4.14. Реализация поиска в мультисписке

```
type
    stype = array[1..20] of char;
    ctype = array[1..5] of char;
    recordkinds = (student, course, enrollment);
    recordtype = record
        case kind : recordkinds of
            student: (studentname: stype;
                      firstcourse: ↑recordtype);
            course:(coursename: ctype;
                      firststudent: ↑recordtype);
            enrollment: (cnext, snext: trecordtype)
        end;

procedure printstudents ( cname: ctype );
    var
        c, e, f: trecordtype;
    begin
        c:= указатель на запись курса, где c↑.coursename = cname;
        { последний оператор зависит от того,
            как реализовано множество записей курсов }
        e:= c↑.firststudent;
        { е пробегает по кольцу указателей
            регистрационных записей }
        while e↑.kind = enrollment do begin
            f:= e;
            repeat
                f:= f↑.cnext
            until
```

```
                    f↑.kind = student;
                    { сейчас f — указатель на студента-собственника
                        регистрации et }
                writeln(f↑.studentname);
                e:= e↑.snext
            end
        end; { printstudents }
```

Эффективность двойных структур данных

Часто кажущаяся простота представления множеств или отображений на самом деле оборачивается трудной проблемой выбора подходящей структуры данных. Какая-либо структура данных, представляющая множество, позволяет легко выполнять определенные операторы, но на выполнение других приходится затрачивать значительное время. По-видимому, не существует одной структуры данных, позволяющей всем операторам выполняться быстро и просто. В этом случае решением может стать использование двух или больше структур данных для представления одного и того же множества или отображения.

Предположим, что мы хотим создать "теннисную лестницу" (т.е. квалификационную таблицу), где каждый игрок располагается на своей "ступеньке" (рейтинге). Новые игроки добавляются в самый низ, т.е. на ступеньку с самым большим номером. Игрок может вызвать на поединок игрока, стоящего на более высокой ступени, и если он выиграет матч, то они меняются ступенями. Эту "лестницу" можно рассматривать как абстрактный тип данных, где базовой моделью будет отображение из множества имен игроков (строки символов) в множество номеров ступеней (целые числа 1, 2, ...). Необходимы также три оператора, выполняемые над этим **АТД**.

1. **ADD(*имя*)** добавляет новое имя в квалификационную лестницу на свободную ступеньку с наибольшим номером, т.е. в самый низ лестницы.
2. **CHALLENGE(*имя*)** возвращает имя игрока, стоящего на ступени $i - 1$, если игрок *имя* стоит на ступени i, $i > 1$.
3. **EXCHANGE(i)**[1] меняет местами игроков, стоящих на ступенях i *и* $i - 1$, $i > 1$.

Для реализации описанных данных и их операторов можно использовать массив *LADDER* (Лестница), где ***LADDER*[*i*]** — имя игрока, стоящего на ступени i. Вместе с именем игрока можно хранить и его номер (т.е. номер ступени, на которой он стоит). В этом случае нового игрока можно просто добавить за малое фиксированное время в свободную ячейку массива.

Оператор EXCHANGE реализуется легко — меняются местами два элемента массива. Но выполнение функции CHALLENGE в процессе поиска заданного имени требует времени порядка $O(n)$ для просмотра всего массива (здесь и далее и — общее число игроков).

С другой стороны, мы можем применить хеш-таблицу для задания отображения из множества имен в множество номеров ступеней. Количество сегментов в хеш-таблице примерно равно количеству игроков. Оператор ADD выполняется в среднем за время $O(1)$. Функция CHALLENGE тратит в среднем $O(1)$ времени на поиск заданного имени, но требует $O(n)$ времени для поиска имени соперника, стоящего на более высокой ступени, поскольку в этом случае надо просмотреть всю хеш-таблицу. Оператор EXCHANGE требует $O(n)$ времени для поиска имен игроков, стоящих на ступенях i и $i - 1$.

Теперь рассмотрим применение комбинации двух структур. Пусть ячейки хеш-таблицы содержат имя игрока и его номер ступени, а в массиве *LADDER* элемент ***LADDER*[*i*]** будет указателем на ячейку игрока в хеш-таблице, стоящего на ***i*-й** ступени, как показано на рис. **4.10.**

---

[1] **Названия** операторов переводятся соответственно как "Добавить", "Вызов" (на поединок) и "Обмен". — *Прим. перев.*

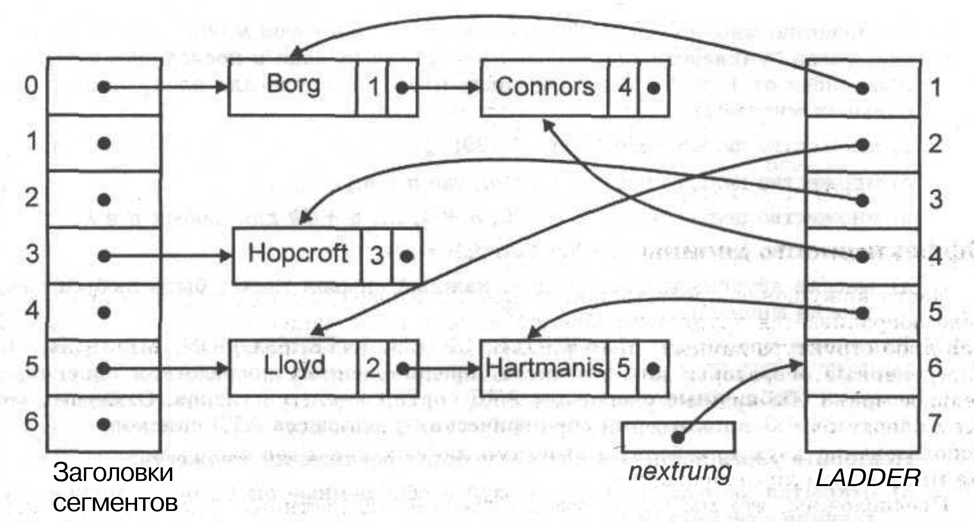


Рис. 4.10. Комбинированная структура высокой производительности

При такой организации данных добавление нового игрока требует в среднем $O(1)$ времени для записи его атрибутов в хеш-таблицу, а также создания указателя на его ячейку в хеш-таблице из ячейки массива *LADDER,* которая помечена курсором *nextrung* (следующая ступень) (см. рис. 4.10). При выполнении функции CHALLENGE надо сначала найти заданное имя в хеш-таблице (на это требуется в среднем $O(1)$ времени), получить его номер ступени i и далее, следуя за указателем из ячейки $LADDER[i - 1]$ в ячейку хеш-таблицы, получить имя игрока-соперника. Все последние описанные действия в среднем требуют не более $O(1)$ времени, поэтому все время выполнения функции CHALLENGE имеет в среднем порядок $O(1)$.

Оператор EXCHANGE(i) требует времени порядка $O(1)$ для нахождения имен игроков с рангами i и i - 1, для обмена содержимым двух ячеек хеш-таблицы и обмена указателями в двух ячейках массива *LADDER.* Таким образом, даже в самом худшем случае время выполнения оператора EXCHANGE не будет превышать некоторой фиксированной величины.

Упражнения

4.1. Пусть заданы два множества $A = \{1, 2, 3\}$ и $B = \{3, 4, 5\}$. Каков будет результат выполнения следующих операторов?

а) UNION(A, *B, C);*

б) INTERSECTION(*A*, Б, C);

в) DIFFERENCE(*A, B. C);*

г) MEMBER(1, *A*);

д) INSERT(1, A);

е) DELETE(1, *A*);

ж) MIN(*A*).

*4.2. Напишите процедуру в терминах базовых операторов множеств для печати всех элементов произвольного конечного множества. Предполагается, что элементы множества имеют тип, допустимый для печати. Процедура печати не должна разрушать исходное множество. Какова в данном случае наиболее подходящая структура данных для реализации множества?

4.3. Реализацию множеств посредством двоичных векторов можно использовать тогда, когда "универсальное множество" преобразовано в последовательность целых чисел от 1 до N. Опишите такое преобразование для следующих универсальных множеств:

а) множество целых чисел О, 1, ..., 90;

б) множество целых чисел от n до m, где га < от;

в) множество целых чисел n, $n + 2$, $n + 4$, ..., га + $2k$ для любых п и fe;

г) множество символов 'a', 'b', ..., V;

д) массив двухсимвольных строк, каждый символ может быть любым символом из множества 'a', 'b', ..., V.

4.4. Напишите процедуры MAKENULL, UNION, INTERSECTION, MEMBER, MIN, INSERT и DELETE для множеств, представленных посредством списков, используя обобщенные операторы АТД сортированных списков. Отметим, что в листинге 4.3 используется специфическая реализация АТД списков.

4.5. Повторите упражнение 4.4 для следующих реализаций множеств:

а) открытая хеш-таблица (используйте обобщенные операторы списков, работающие с сегментами);

б) закрытая хеш-таблица с линейной методикой разрешения коллизий;

в) несортированный список (используйте обобщенные операторы списков);

г) массив фиксированной длины и указатель на последнюю занятую позицию в массиве.

4.6. Для каждой реализации и каждого оператора из упражнений 4.4 и 4.5 найдите порядок времени выполнения операторов над множествами размера n.

4.7. Предположим, что для хеширования целых чисел в 7-сегментную хеш-таблицу используется хеш-функция $h(i)$ = i mod 7.

а) приведите результирующую хеш-таблицу, если в нее вставляются точные кубы 1, 8, 27, 64, 125, 216, 343;

б) повторите предыдущий пункт для закрытой хеш-таблицы с линейной методикой разрешения коллизий.

4.8. Предположим, что есть закрытая хеш-таблица с 5 сегментами, хеш-функцией $h(i)$ = i mod 5 и линейной методикой разрешения коллизий. Покажите результат вставки в первоначально пустую хеш-таблицу последовательности чисел 23, 48, 35, 4, 10.

4.9. Приведите реализацию операторов АТД отображений с использованием открытой и закрытой хеш-таблиц.

4.10. Чтобы увеличить скорость выполнения операторов, мы хотим заменить открытую хеш-таблицу с B_1 сегментами, содержащую значительно больше чем B_1 элементов, на другую хеш-таблицу с B_2 сегментами. Напишите процедуру преобразования старой хеш-таблицы в новую, используя операторы АТД списков для заполнения каждого сегмента.

4.11. В разделе 4.8 мы обсуждали "случайные" хеш-функции, когда для проверки сегментов после возникновения i-й коллизии применяется функция $h_i(x) = (h(x) + d_i) \bmod B$ с использованием некоторой последовательности d_1, $d_2, \ldots, d_{B-1}$. Мы также показали один способ вычисления этой последовательности, когда задается некоторая константа k, первое значение последовательности, обычно $d_1 > 0$, и применяется формула

$$d_i = \begin{cases} \quad {}_{i-1}, & \text{если } 2d_{i-1} < B, \\ (2d_{i-1} - B) \oplus \text{ft}, & \text{если } 2d_{i-1} > B, \end{cases}$$

где $i > 1$, B является степенью 2, а знак $\oplus$ обозначает сложение по модулю 2. Для случая $B = 16$ найдите такое значение k, чтобы последовательность $d_1, d_2, ..., d_{15}$ включала все целые числа 1, 2, ..., 15.

4.12. Нарисуйте частично упорядоченное дерево, полученное в результате вставки в пустое дерево целых чисел 5, 6, 4, 9, 3, 1 и 7. Каков будет результат последовательного применения к этому дереву трех операторов DELETEMIN?

4.13. Предположим, что множество учебных курсов (см. пример из раздела 4.12) представлено в виде

а) связанного списка;

б) хеш-таблицы;

в) дерева двоичного поиска.

Измените объявление типов в листинге 4.14 для каждой из этих структур.

4.14. Измените структуру данных в листинге 4.14 так, чтобы каждая регистрационная запись непосредственно указывала на своих собственников среди записей о студентах и учебных курсах. Перепишите процедуру ***printstudents*** из листинга 4.14 для этой структуры данных.

4.15. Предположим, что есть 20 000 студентов, 1 000 учебных курсов и что каждый студент записан в среднем на три учебных курса. Сравните структуру данных из листинга 4.14 с измененной структурой упражнения 4.14 по следующим показателям:

а) по требуемому пространству памяти;

б) по среднему времени выполнения процедуры *printstudents;*

в) по среднему времени выполнения процедуры печати названий курсов, аналогичной процедуре *printstudents.*

4.16. Для структуры данных из листинга 4.14 напишите процедуры вставки и удаления отношения "студент *s* посещает курс с".

4.17. Каковы различия (если они есть) между структурой данных, представленной в листинге 4.14, и структурой, в которой множества C_s и S_c представлены списками указателей на соответствующие записи студентов и курсов.

4.18. Работники некой компании представлены в базе данных этой компании своими именами (предполагается, что все они уникальны), табельными номерами и номерами социального страхования. Основываясь на этих предположениях, предложите структуру данных, позволяющую найти повторяющиеся записи одного и того же работника. Как быстро (в среднем) можно выполнить такую операцию?

Библиографические примечания

Монография [67] является прекрасным источником дополнительной информации о хешировании. Методы хеширования начали развиваться с середины 50-х годов, и статья [85] — фундаментальная ранняя работа в этой области. Работы [73] и [77] содержат хорошие обзоры методов хеширования.

Мультисписки, как основная структура данных для сетевых распределенных баз данных, предложены в [22]. В работе [112] имеется дополнительная информация о применении таких структур в системах баз данных.

Реализация частично упорядоченных деревьев посредством куч — основная идея работы [119]. Очереди с приоритетами ранее рассматривались в книге [65].

В [89] обсуждается вычислительная сложность основных операторов множеств. Техника анализа потоков данных, основанных на множествах, подробно рассмотрена в работах [5] и [18].

ГЛАВА 5

Специальные методы представления множеств

В этой главе мы рассмотрим структуры данных для представления множеств, которые позволяют более эффективно реализовать общий набор операторов, выполняемых над множествами, чем это описано в предыдущих главах. Однако эти структуры значительно сложнее и чаще всего применяются только для больших множеств. Все они основаны на различного рода деревьях, таких как деревья двоичного поиска, нагруженные и сбалансированные деревья.

5.1. Деревья двоичного поиска

Начнем с деревьев двоичного поиска — основной структуры данных для представления множеств, чьи элементы упорядочены посредством некоторого отношения линейного порядка, которые, как обычно, обозначим символом "<". Эти структуры особенно полезны, когда исходное множество такое большое, что не рационально использовать его элементы непосредственно в качестве индексов массивов. Предполагается, что все элементы множеств являются элементами некоторого универсального множества — универсума, примером которого служит множество возможных идентификаторов в программе на языке Pascal. На деревьях двоичного поиска можно реализовать операторы INSERT, DELETE, MEMBER и MIN, причем время их выполнения в среднем имеет порядок $O(\log n)$ для множеств, состоящих из n элементов.

Дерево двоичного поиска — это двоичное дерево, узлы которого помечены элементами множеств (мы также будем говорить, что узлы дерева содержат или хранят элементы множества). Определяющее свойство дерева двоичного поиска заключается в том, что все элементы, хранящиеся в узлах левого поддерева любого узла x, меньше элемента, содержащегося в узле x, а все элементы, хранящиеся в узлах правого поддерева узла x, больше элемента, содержащегося в узле x. Это свойство называется *характеристическим свойством дерева двоичного поиска* и выполняется для любого узла дерева двоичного поиска, включая его корень.

На рис. 5.1 показаны два дерева двоичного поиска, представляющие одно и то же множество целых чисел. Отметим интересное свойство деревьев двоичного поиска: если составить список узлов этого дерева, обходя его во внутреннем (симметричном) порядке, то полученный список будет отсортирован в порядке возрастания (в соответствии с заданным на множестве отношением линейного порядка).

Пусть дерево двоичного поиска представляет некоторое множество элементов. Характеристическое свойство дерева двоичного поиска позволяет легко проверить принадлежность любого элемента исходному множеству. Для того чтобы определить, принадлежит ли элемент x исходному множеству, сначала сравним его с элементом r, находящимся в корне дерева. Если $x = r$, то вопрос о принадлежности элемента x множеству решен положительно. В случае $x < r$ по характеристическому свойству дерева двоичного поиска

элемент x может быть потомком только левого сына корня дерева[1]. Аналогично, если $x > r$, то элемент x может быть потомком только Правого сына корня дерева.

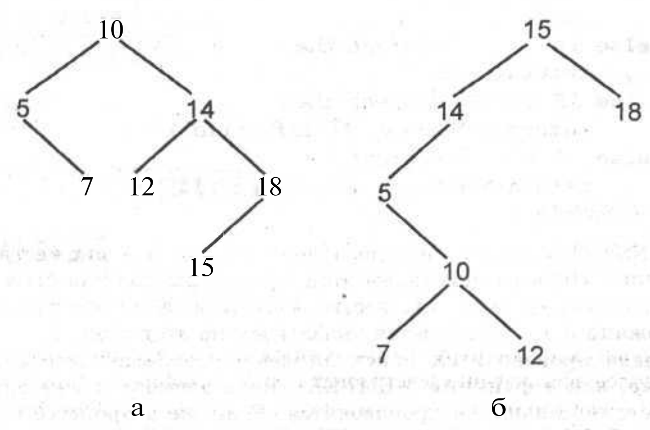


Рис. 5.1. Два дерева двоичного поиска

Напишем простую рекурсивную функцию **MEMBER**(x, A), реализующую тест на принадлежность элемента множеству. Предположим, что элементы множества имеют пока не определенный тип данных, который назовем elementtype. Будем считать, что для этого типа данных определены отношения "<" и "=". Если это не так, то необходимо написать функции **LT**(a, b) и **EQ**(a, b) такие, что для любых элементов а и b типа elementtype функция **LT**(a, b) будет принимать значение true тогда и только тогда, когда а "меньше" b, a функция **EQ**(a, b) — когда a "равно или больше" b.

Тип данных nodetype узлов дерева, содержащих поле *element* для элементов множества и два поля *leftchild* и *rightchild* указателей на левого и правого сыновей, определим с помощью следующего объявления:

```
type
    nodetype = record
        element: elementtype;
        leftchild, rightchild: ↑nodetype
    end;
```

Далее определим тип SET (Множество) как указатель на узел, который будет корнем дерева двоичного поиска при представлении множества:

```
type
    SET: ↑nodetype;
```

Теперь можно написать полный код функции MEMBER (листинг 5.1). Отметим, что, поскольку в данном случае SET и указатель на узел — синонимы, функция может вызывать сама себя для проведения поиска по поддереву, как если бы это поддерево представляло все множество. Другими словами, множество можно как бы разбить на подмножество элементов, которые меньше x, и подмножество элементов, больших x.

Листинг **5.1. Процедура** MEMBER для дерева двоичного **поиска**

```
function MEMBER ( x: elementtype; A: SET ): boolean;
    { возвращает true, если элемент x принадлежит множеству A,
      false — в противном случае }
```

[1] **Напомним, что любой узел считается (по определению из раздела 3.1 главы 3) предком самого себя, поэтому мы здесь не исключаем возможность, что элемент *x* может совпасть с левым сыном корня.**

```
    begin
        if A = nil then
            return(false) { x не может принадлежать 0 }
        else if x = A↑.element then
            return(true)
        else if x < A↑.element then
            return(MEMBER(x, A↑.leftchild))
        else  { x > A↑.element }
            return(MEMBER(x, A↑.rightchild))
    end; { MEMBER }
```

Процедура INSERT(*x*, *A*), которая добавляет элемент *x* в множество *A*, также проста для написания. Первым действием этой процедуры должна быть проверка условия *A* = nil, т.е. не является ли множество *A* пустым. Если это так, то создается новый узел, содержащий *x*, и *A* делается указателем на этот узел. Если множество *A* не пустое, то, сначала производится поиск элемента, совпадающего с *x* (это делается примерно так же, как в функции MEMBER). Если элемент *x* уже есть в множестве, то никакие действия дальше не производятся. Если же в процессе поиска будет достигнут указатель **nil** (т.е. дошли до листа дерева), то он заменяется на указатель, указывающий на новый узел, содержащий элемент *x*. Код этой процедуры приведен в листинге 5.2.

Листинг 5.2. Вставка нового элемента в дерево двоичного поиска

```
procedure INSERT ( x: elementtype; var A: SET );
    begin
        if A - nil then begin
            new(A);
            A↑.element:= x;
            A↑.leftchild:= nil;
            A↑.rightchild:= nil
        end
        else if x < A↑.element then
            INSERT(x, A↑.leftchild)
        else if x > A↑.element then
            INSERT(x, A↑.rightchild)
        { если x = A↑.element, то никаких действий
            не производится, т.к. x уже есть в множестве A }
    end; { INSERT }
```

Удаление элемента представляет некоторые трудности. Сначала надо найти этот элемент в дереве. Если *x* является листом, то он просто удаляется. Но если *x* является внутренним узлом (назовем его для краткости ***inode***, от interior node — внутренний узел), то его нельзя просто удалить, так как нарушится связанность дерева.

Если *inode* имеет только одного сына (как узел 14 на рис. 5.1,б), то его можно заменить этим сыном. Если *inode* имеет двух сыновей, как узел 10 на рис. 5.1,а, то среди потомков правого сына надо найти элемент с наименьшим значением[1] и заменить им удаляемый элемент. Например, для узла 10 на рис. 5.1,а таким будет узел 12.

Для написания процедуры DELETE полезно иметь функцию DELETEMIN(*A*), которая удаляет наименьший элемент из непустого дерева и возвращает значение удаляемого элемента. Код функции DELETEMIN приведен в листинге *5.3, а* код процедуры DELETE, использующий эту функцию, — в листинге 5.4.

---

[1] Можно также искать элемент с наибольшим значением среди потомков левого сына.

Листинг 5.3. Удаление наименьшего элемента

```
function DELETEMIN ( var A: SET ): elementtype;
    begin
        if A↑.leftchild = nil then begin
            { A указывает на наименьший элемент }
            DELETEMIN:= A↑.element;
            A:= A↑.rightchild
                { замена узла, указанного A, его правым сыном }
        end
        else { узел, указанный A, имеет левого сына }
            DELETEMIN:= DELETEMIN(A↑.leftchild)
    end; { DELETEMIN }
```

Листинг 5.4. Удаление элемента из дерева двоичного поиска

```
procedure DELETE ( x: elementtype; var A: SET );
begin
    if A<> nil then
        if x < A↑.element then
            DELETE(x, A↑.leftchild)
        else if x > A↑.element then
            DELETE(x, A↑.rightchild)
        else if (A↑.leftchild= nil) and (A↑.rightchild= nil) then
            A:= nil { удаление листа, содержащего x }
        else if A↑.leftchild = nil then
            A:= A↑.rightchild
        else if A↑.rightchild = nil then
            A:= A↑.leftchild
        else { у узла есть оба сына }
            A↑.element:= DELETEMIN(A↑.rightchild)
end; { DELETE }
```

Пример 5.1. Предположим, надо удалить элемент 10 из дерева, показанного на рис. 5.1,а. В данном случае первым исполняемым оператором в процедуре DELETE будет вызов функции DELETEMIN с аргументом-указателем на узел 14. Этот указатель — значение поля *rightchild* корня. Результатом этого вызова будет еще один вызов той же функции DELETEMIN. В этом случае ее аргументом будет указатель на узел 12, который находится в поле *leftchild* узла 14. Узел 12 не имеет левого сына, поэтому функция возвращает элемент 12 и устанавливает указатель узла 14 на левого сына в состояние nil. Затем процедура DELETE берет значение 12, возвращенное функцией DELETEMIN, и заменяет им элемент 10. Результирующее дерево показано на рис. 5.2. □

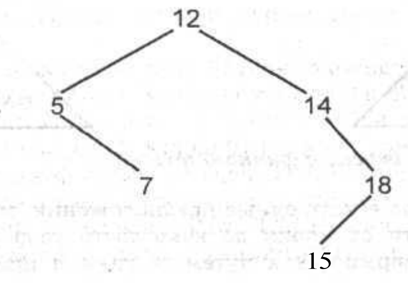


Рис. 5.2. Дерево, представленное на рис. 5.1,а, после удаления элемента 10

5.2. Анализ времени выполнения операторов

В этом разделе мы проанализируем работу различных операторов, выполняемых над деревьями двоичного поиска. Мы покажем, что если в первоначально пустое дерево двоичного поиска вставлено n произвольных элементов, то средняя длина пути от корня до листа будет $O(\log n)$. Отсюда следует, что среднее время выполнения оператора MEMBER также будет иметь порядок $O(\log n)$.

Легко видеть, что если двоичное дерево полное (т.е. все узлы, за исключением самого нижнего уровня, имеют двух сыновей), то не существует пути, содержащего более $1 + \log n$ узлов[1]. Поэтому операторы MEMBER, INSERT, DELETE и DELETEMIN имеют время выполнения порядка $O(\log n)$. Чтобы доказать это, достаточно заметить, что все эти операторы затрачивают фиксированное время на обработку одного узла, затем рекурсивно вызывают себя для обработки сына рассматриваемого узла. Последовательность узлов, в которых выполняется рекурсивный вызов процедур, формирует путь, исходящий из корня дерева. Поскольку такой путь имеет в среднем длину $O(\log n)$, то и общее время прохождения операторами этого пути будет иметь такой же порядок.

Однако когда элементы вставляются в произвольном порядке, они не обязаны располагаться в виде полного двоичного дерева. Например, если вставка начата с наименьшего элемента и далее элементы вставляются в порядке возрастания, то получим дерево в виде цепочки узлов, где каждый узел имеет правого сына, но не имеет левого. В этом случае, как легко показать, на вставку i-го элемента требуется O(i) шагов, для завершения всего процесса вставки га элементов необходимо $\sum_{i=1} i = n(n + 1)/2$ шагов, т.е. порядка $O(n^2)$, или $O(n)$ на одно выполнение оператора вставки.

Мы должны уметь определять, когда "среднее" дерево двоичного поиска из n узлов ближе по своей структуре к полному двоичному дереву, а когда к цепочке узлов, так как в первом случае среднее время выполнения операторов имеет порядок $O(\log n)$, а во втором — $O(n)$. Если мы не знаем последовательность выполненных (или тех, которые будут выполнены в будущем) операций вставки и удаления элементов, а также свойства этих элементов (например, мы не можем всегда удалять только минимальные элементы), то естественно предположить, что анализу подлежат только пути "средней" длины на "случайных" деревьях. В частности, мы вправе предполагать, что деревья сформированы только вставкой элементов и все последовательности вставляемых элементов равновероятны на фиксированном множестве из п элементов.

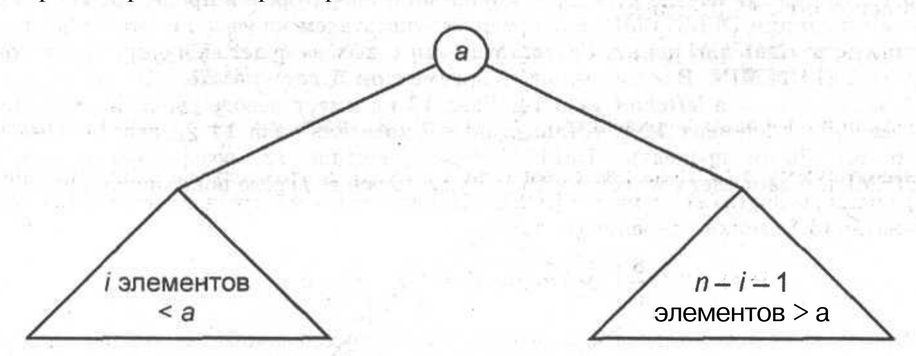


Рис. 5.3. Построение дерева двоичного поиска

Принимая эти достаточно естественные предположения, можно вычислить $P(n)$ — среднее число узлов в пути от корня до некоторого узла (не обязательно листа). Предполагая, что дерево формируется путем вставки n произвольных элементов в

[1] Напомним, что все логарифмы, пока не сказано другое, определены по основанию 2.

первоначально пустое дерево, очевидно, что $P(0) = 0$ и $P(1) = 1$. Пусть в списке из $n \geq 2$ элементов, готовых к вставке в пустое дерево, первым находится элемент a. Если отсортировать список, то элемент a с равной вероятностью может занимать любое место от 1 до n. Пусть в списке i элементов меньше a, и, следовательно, $n - i - 1$ элементов больше, чем a. Тогда при построении дерева в его корне поместится элемент a (поскольку он первый элемент списка), i наименьших элементов будут потомками левого сына корня, а оставшиеся $n - i - 1$ элементов — потомками правого сына корня. Это дерево показано на рис. 5.3.

Так как любые последовательности i наименьших элементов и $n - i - 1$ наибольших элементов равновероятны, мы вправе предположить, что средняя длина путей в левом и правом поддеревьях будет $P(i)$ и $P(n - i - 1)$ соответственно. Поскольку пути должны начинаться от корня полного дерева, то к $P(i)$ и $P(n - i - 1)$ надо еще прибавить 1. Тогда $P(n)$ можно вычислить путем усреднения для всех i от 0 до $n - 1$ следующих сумм:

$$\frac{i}{n}(P(i)+1) + \frac{n-i-1}{n}(P(n-i-1)+1) + \frac{1}{n},$$

где первое слагаемое — средняя длина пути в левом поддереве с весовым коэффициентом i/n, значение второго слагаемого аналогично, а $1/n$ соответствует "вкладу" в путь корня. Усредняя эти выражения по всем i от 0 до $n - 1$, получим рекуррентное уравнение

$$P(n) = 1 + \frac{1}{n^2}\sum_{i=0}^{n-1}\left(iP(i) + (n-i-1)P(n-i-1)\right). \tag{5.1}$$

Вторая часть этой суммы $\sum_{i=0}^{n-1}(n-i-1)P(n-i-1)$, если подставить i вместо $(n - i - 1)$, преобразуется в первую часть (5.1) $\sum_{i=0}^{n-1} iP(i)$. В последней сумме слагаемое при $i = 0$ равно нулю, поэтому суммирование можно начинать с $i = 1$. Таким образом, формулу (5.1) можно переписать в следующем виде:

$$P(n) = 1 + \frac{2}{n^2}\sum_{i=1}^{n-1} iP(i), \quad n \geq 2. \tag{5.2}$$

Теперь индукцией по n покажем, что $P(n) \leq 1 + 4\log n$. Очевидно, что это утверждение справедливо для $n = 1$, поскольку $P(1) = 1$, и для $n = 2$, так как $P(2) = 2 \leq 1 + 4\log 2 = 5$. Предположим, что оно выполняется для всех $i < n$. Тогда на основании (5.2) имеем

$$P(n) \leq 1 + \frac{2}{n^2}\sum_{i=1}^{n-1}(4i\log i + i) = 1 + \frac{2}{n^2}\sum_{i=1}^{n-1} 4i\log i + \frac{2}{n^2}\sum_{i=1}^{n-1} i \leq 2 + \frac{8}{n^2}\sum_{i=1}^{n-1} i\log i. \tag{5.3}$$

Здесь использовано неравенство $\sum_{i=1}^{n-1} i \leq n^2/2$, поэтому $\frac{2}{n^2}\sum_{i=1}^{n-1} i \leq 1$. Разделим последнюю сумму в (5.3) на две суммы: по $i \leq \lceil n/2 \rceil$, в этом случае слагаемые не будут превышать $i\log(n/2)$, и по $i > \lceil n/2 \rceil$, где слагаемые не превышают $i\log(n)$. Тогда неравенство (5.3) можно переписать так:

$$P(n) \leq 2 + \frac{8}{n^2}\left(\sum_{i=1}^{\lceil n/2 \rceil} i\log(n/2) + \sum_{i=\lceil n/2 \rceil+1}^{n-1} i\log n\right). \tag{5.4}$$

Независимо от того, четное или нечетное n, нетрудно показать, что первая сумма в (5.4) не превышает величины $(n^2/8)\log(n/2)$, которая равна $(n^2/8)\log n - (n^2/8)$, а вторая не превышает $(3n^2/8)\log n$. С учетом этих оценок из (5.4) получаем

$$P(n) \leq 2 + \frac{8}{n^2}\left(\frac{n^2}{2}\log n - \frac{n^2}{8}\right) \leq 1 + 4\log n.$$

Этот шаг завершает индукцию и доказывает утверждение. Итак, доказано, что в дереве двоичного поиска, которое получено путем случайной вставки n элементов, средняя

длина пути от корня до произвольного узла имеет порядок $O(\log n)$, т.е. точно такой же порядок (с точностью до константы пропорциональности), как и для полного двоичного дерева. Более точный анализ показывает, что константу 4 в вышеприведенной оценке можно заменить на 1.4.

Из доказанного утверждения следует, что время выполнения проверки на принадлежность произвольного элемента к множеству имеет порядок $O(\log n)$. Подобный анализ показывает, что если мы включим в путь только те узлы, которые имеют обоих сыновей, или только узлы, имеющие лишь левого сына, то все равно для средней длины пути получим уравнение, подобное (5.1), и поэтому будет оценка порядка $O(\log n)$. Отсюда следует вывод, что для выполнения операций проверки на принадлежность к множеству элемента, которого заведомо нет в множестве, вставки произвольного нового элемента или удаления элемента также требуется в среднем времени порядка $O(\log n)$.

Эффективность деревьев двоичного поиска

При реализации словарей посредством **хеш-таблиц** время выполнения основных операторов в среднем постоянно, т.е. практически не зависит от размера множеств. Поэтому такая реализация более эффективна, чем реализация деревьев двоичного поиска, за исключением того случая, когда необходимо часто применять оператор MIN, который требует времени порядка $O(n)$ для хеш-таблиц.

Сравним деревья двоичного поиска с частично упорядоченными деревьями, которые применяются для реализации очередей с приоритетами (см. главу 4). Частично упорядоченные деревья из n элементов требуют только $O(\log n)$ шагов для выполнения операторов INSERT и DELETEMIN, но не в среднем, а в самом худшем случае. Более того, константа пропорциональности перед $\log n$ для частично упорядоченных деревьев меньше, чем для деревьев двоичного поиска. Но на дереве двоичного поиска могут выполняться как операторы DELETE и MIN, так и их комбинация DELETEMIN, тогда как на частично упорядоченном дереве выполняется только последний из этих операторов. Кроме того, оператор MEMBER требует $O(n)$ шагов на частично упорядоченном дереве, но только $O(\log n)$ шагов на дереве двоичного поиска. Таким образом, частично упорядоченные деревья хорошо подходят для реализации очередей с приоритетами, но не эффективны при выполнении других операторов множеств, которые могут выполняться на деревьях двоичного поиска.

5.3. Нагруженные деревья

В этом разделе мы рассмотрим специальную структуру для представления множеств, состоящих из символьных строк. Некоторые из описанных здесь методов могут работать и со строками объектов другого типа, например со строками целых чисел. Эта **структура** называется *нагруженными деревьями*[1] (tries). Рассмотрим их применение к символьным строкам.

---

[1] В оригинале структура называется *trie,* это слово получено из букв, стоящих в середине слова "retrieval" (поиск, выборка, возврат). Устоявшегося термина для этой структуры в русской литературе пока нет. (О расхождении терминологии можно судить по переводу известной книги *D.E. Knuth. The art of computer programming, Vol. III: Sorting and Searching:* в первом русском переводе (Кнут Д. *Искусство программирования для ЭВМ. Т. 3: Сортировка и поиск.* — М., "Мир", 1978 г.) вместо *trie* используется термин *бор* (от слова выборка), в последнем переводе переработанного издания этой книги (Кнут Д. *Искусство программирования. Т. 3: Сортировка и поиск.* — М., Издательский дом "Вильяме", 2000 г.) — термин *луч* (от слова **получение**).) Чтобы не заниматься "терминотворчеством", мы применили известный термин *нагруженные деревья,* который соответствует смыслу этой структуры. В данном случае можно было бы применить и термин *синтаксические деревья,* но, хотя этот термин по своему значению и "пересекается" с термином *нагруженные деревья,* он имеет другую направленность. — *Прим. ред.*

Пример 5.2. Как описывалось в главе 1, возможная реализация проверки орфографии состоит из следующей последовательности действий: сначала читается слово из текстового файла с остановкой после каждого слова (слова отделяются друг от друга пробелом или символом новой строки), далее проверяется, есть ли это слово в стандартном словаре наиболее употребительных слов. Слова, отсутствующие в словаре, распечатываются как возможные ошибки. В листинге 5.5 приведен эскиз возможной программы *spell* (правописание). Здесь использована процедура *getword(x, f)*, которая читает следующее слово в текстовом файле *f* и присваивает его переменной *x*, имеющей тип wordtype (мы определим его позже). Еще одна используемая переменная А имеет знакомый нам тип SET. Реализация этого АТД использует операторы INSERT, DELETE, MAKENULL и PRINT. Последний оператор распечатывает элементы множества. □

Листинг 5.5. Программа проверки орфографии

```
program spell ( input, output, dictionary );
    type
        wordtype = { надо определить }
        SET = { надо определить, используя структуру
                                        нагруженного дерева }
    var
        A: SET; { хранит считанное слово, пока оно
                                        не найдено в словаре }
        nextword: wordtype;
        dictionary: file of char;

    procedure getword ( var x: wordtype; f: file of char );
        { читает следующее слово в файле f и
                присваивает его переменной x }

    procedure INSERT ( x: wordtype; var S: SET ) ;
        { надо определить }

    procedure DELETE ( x: wordtype; var S: SET );
        { надо определить }

    procedure MAKENULL ( var S: SET ) ;
        { надо определить }

    procedure PRINT ( var S: SET );
        { надо определить }
    begin
        MAKENULL(A);
        while not eof(input) do begin
            getword(nextword, input);
            INSERT(nextword, A)
        end;
        while not eof(dictionary) do begin
            getword(nextword,dictionary) -,
            DELETE(nextword, A)
        end;
        PRINT(A)
    end; { spell }
```

Структура нагруженных деревьев поддерживает операторы множеств, у которых элементы являются словами, т.е. символьными строками. Удобно, когда

большинство слов начинается с одной и той же последовательности букв или, другими словами, когда количество различных префиксов значительно меньше общей длины всех слов множества.

В нагруженном дереве каждый путь от корня к листу соответствует одному слову из множества. При таком подходе узлы дерева соответствуют префиксам слов множества. Чтобы избежать коллизий слов, подобных THE и THEN, введем специальный символ $, *маркер конца,* **указывающий** окончание любого слова. Тогда только слова будут словами, но не префиксы.

Пример 5.3. На рис. 5.4 показано нагруженное дерево слов {THE, THEN, THIN, THIS, TIN, SIN, SING}. Здесь корень соответствует пустой строке, а два его сына — префиксам T и S. Самый левый лист представляет слово THE, следующий лист — слово THEN и т.д. □

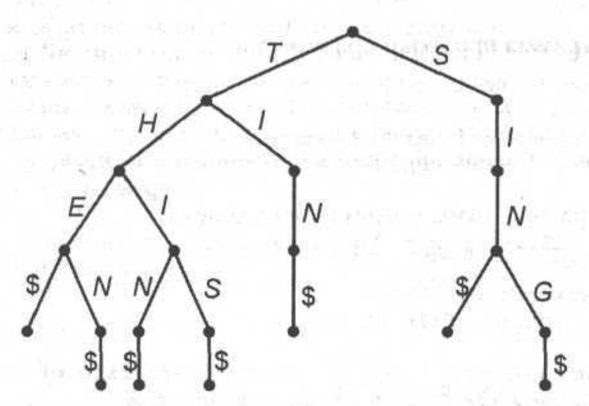


Рис. 5.4. Нагруженное дерево

На основании рис. 5.4 можно сделать следующие выводы.

1. Каждый узел может иметь до 27 сыновей[1], которые могут быть буквами или символом $.
2. Большинство узлов имеет значительно меньше 27 сыновей.
3. Листом может быть только окончание ребра, помеченного символом $.

Узлы нагруженного дерева как АТД

Мы можем рассматривать узел нагруженного дерева как отображение, где областью определения будет множество {A, B, ..., Z, $} (или другой выбранный алфавит), а множеством значений — множество элементов типа "указатель на узел нагруженного дерева". Более того, так как дерево можно идентифицировать посредством его корня, то АТД TRIE (Нагруженное дерево) и TRIENODE (Узел нагруженного дерева) имеют один и тот же тип данных, хотя операторы, используемые в рамках этих АТД, имеют существенные различия. Для реализации АТД TRIENODE необходимы следующие операторы.

1. Процедура ASSIGN(*node, c, p),* которая задает значение *p* (указатель на узел) символу *c* в узле *node,*
2. Функция VALUEOF(*node, c*)[2] — возвращает значение (указатель на узел), ассоциированное с символом *c в* узле *node,*

---

[1] Здесь предполагается, что алгоритм работает только с английским алфавитом. — *Прим. ред.*

[2] Эта функция является версией функции COMPUTER из раздела 2.5.

3. Процедура **GETNEW(*node*,** c) делает значение узла *node c* символом с указателем на новый узел.

Нам также будет необходима процедура **MAKENULL(*node*),** делающая узел *node* пустым отображением.

Самой простой реализацией узлов нагруженного дерева будет массив *node* указателей на узлы с индексным множеством {**A,** B, ..., Z, $}. Таким образом, АТД TRIENODE можно определить следующим образом:

```
type
    chars = {'A', 'B', ..., 'Z', '$'};
    TRIENODE = array[chars] of T TRIENODE;
```

Если *node* — массив узлов, то *node[c]* совпадает с **VALUEOF(*node*,** c) для любого символа *c*. Чтобы избежать создания многочисленных листьев, являющихся потомками '$', примем соглашение, что ***node*['$']** имеет значение **nil** или указателя на самого себя. (При формировании дерева *node* не может иметь сына, соответствующего '$', но после возможных преобразований может появиться такой сын, которого мы никогда не создавали.) Теперь можно написать процедуры, выполняемые над узлами нагруженного дерева. Их код приведен в следующем листинге.

Листинг 5.6. Операторы узлов нагруженного дерева

```
procedure MAKENULL ( var node: TRIENODE );
    { делает node листом, т.е. пустым отображением }
    var
        c: chars;
    begin
        for c:= 'A' to '$' do
            node[c]:= nil
    end; { MAKENULL }

procedure ASSIGN { var node: TRIENODE; c: chars; p: ↑TRIENODE );
    begin
        node[c]:= p
    end; { ASSIGN }

function VALUEOF ( var node: TRIENODE; c: chars ): tTRIENODE;
    begin
        return(node[c])
    end; { VALUEOF }

procedure GETNEW ( var node: TRIENODE; c: chars );
    begin
        new(node[c]);
        MAKENULL(node[c])
    end; { GETNEW }
```

Теперь определим АТД TRIE:

```
type
    TRIE = tTRIENODE;
```

Мы можем определить тип wordtype как массив символов фиксированной длины. **Предполагается,** что среди значений такого массива обязательно есть по крайней мере один символ '$'. Считаем, что концу слова соответствует первый символ '$', независимо от того, что за ним следует. При выполнении этих предположений написана процедура **INSERT(*x*, *words*),** которая вставляет слово *x* в множество *words* (слова), представленное **посредством** нагруженного дерева. Код этой процедуры показан в

листинге 5.7. Написание процедур MAKENULL, DELETE и PRINT для данной реализации нагруженных деревьев оставляем в качестве упражнений для читателей.

Листинг 5.7. Процедура INSERT

```
procedure INSERT ( x: wordtype; var words: TRIE );
    var
        i: integer; { считает позиции в слове x }
        t: TRIE; { используется для указания на узлы дерева,
                       соответствующие префиксу слова x }
    begin
        i:= 1;
        t:= words;
        while x[i] <> '$' do begin
            if VALUEOF(t↑, x[i]) = nil then
                { если текущий узел не имеет сына для
                    символа x[i], то он создается }
                GETNEW(t↑, x[i]);
            t:= VALUEOF(t↑, x[i]);
                { продвижение к сыну узла t для символа x[i] }
            i:= i + 1 { перемещение далее по слову x }
        end;
        { в слове x достигнут первый символ '$' }
        ASSIGN(t↑, '$', t)
            { делает петлю на '$' для создания листа }
    end; { INSERT }
```

Представление узлов нагруженного дерева посредством списков

Множество слов, которые имеют *p* различных префиксов и для хранения требуют *27p* байт, можно представить посредством реализации узлов нагруженного дерева с помощью массивов фиксированной длины. Величина 27*p* байт может значительно превышать общую длину слов в множестве. Существует другая реализация нагруженных деревьев, которая экономит используемый объем памяти. Напомним, что мы рассматриваем каждый узел нагруженного дерева как отображение (см. раздел 2.6). В принципе, можно применить любую реализацию отображений, но мы хотим найти наиболее подходящую, область определения которых сравнительно мала. При таком условии наилучшим выбором будет реализация отображения посредством связанных списков. Здесь представлением отображения, т.е. узла нагруженного дерева, будет связанный список символов, для которых соответствующие значения не являются указателями **nil.** Такое представление можно сделать с помощью следующего объявления:

```
type
    celltype = record
        domain: chars;
        value: ↑celltype;
            { указатель на первую ячейку списка узла сына }
        next: ↑celltype
            { указатель на следующую ячейку списка }
    end;
```

Мы оставляем в качестве упражнений написание для данной реализации процедур ASSIGN, VALUEOF, MAKENULL и GETNEW. После создания этих процедур процедура INSERT из листинга 5.7 должна стать работоспособной программой.

Эффективность структуры данных нагруженных деревьев

Для хеш-таблиц и нагруженных деревьев сравним необходимое время выполнения операторов и занимаемый объем памяти для представления n слов с p различными префиксами и общей диной слов l. Будем предполагать, что указатели требуют для своего хранения по 4 байт на указатель. По всей видимости, наиболее эффективными по критерию требуемой памяти и поддержке операторов INSERT и DELETE будут хеш-таблицы. Если слова имеют разную длину, то ячейки сегментов хеш-таблицы не будут непосредственно содержать слова, а только два указателя. Один требуется для связи ячеек сегмента, а другой будет указывать на начало слова, соответствующего сегменту.

Сами слова хранятся в большом символьном массиве, где начало и конец каждого слова помечается специальным маркером, таким как '$'. Например, слова THE, THEN и THIN будут храниться как

THE$THEN$THIN$ . . .

Указателями для этих слов будут курсоры, указывающие на позиции 1, 5 и 10 массива. Подсчитаем пространство, занимаемое сегментами хеш-таблицы и массивом символов.

1. 8га байт для ячеек сегментов — по одной ячейке на слово. Ячейка содержит два указателя, т.е. занимает 8 байт, которые умножаются на п (количество слов).
2. $l + n$ байт для символьного массива, хранящего n слов общей длиной l и n разделителей слов.

Таким образом, общее занимаемое пространство составляет 9га + l плюс пространство, используемое для заголовков сегментов.

Для сравнения: нагруженное дерево с узлами, представленными связанными списками, требует $p + n$ ячеек — по одной ячейке на каждый префикс и по одной ячейке на каждое окончание слова. Каждая ячейка содержит символ и два указателя, т.е. всего 9 байт. Поэтому общее занимаемое пространство составляет 9p + $9n$ байт. Если l и пространство заголовков сегментов больше $9p$, то нагруженное дерево занимает меньше пространства, чем хеш-таблица. Но в реальных приложениях, таких как словари, отношение l/pобычно меньше 3, поэтому в этих приложениях хеш-таблицы более экономичны.

С другой стороны, к достоинствам нагруженных деревьев можно отнести возможность перемещения по дереву и выполнения таких операторов, как INSERT, DELETE и MEMBER, за время, пропорциональное длине "обслуживаемого" слова. Хеш-функция, чтобы быть действительно "случайной", хеширует каждый символ слова. Поэтому ясно, что вычисления хеш-функции требуют **примерно** такого же времени, как и выполнение, например, оператора MEMBER для нагруженного дерева. И, конечно, время вычисления хеш-функции не включает время, необходимое для разрешения коллизий или выполнения операций вставки, удаления или поиска. Поэтому мы вправе ожидать, что нагруженные деревья будут работать значительно быстрее со словарями, состоящими из символьных строк, чем хеш-таблицы.

Другим достоинством нагруженных деревьев является то, что, в отличие от хеш-таблиц, они поддерживают эффективное выполнение оператора MIN. Кроме того, если хеш-таблица построена так, как описано выше, то после удаления слова из словаря здесь нельзя сравнительно просто организовать повторное использование освободившегося пространства в массиве символов.

5.4. Реализация множеств посредством сбалансированныхдеревьев

В разделах 5.1 и 5.2 мы рассмотрели способы представления множеств посредством деревьев двоичного поиска и узнали, что операторы, подобные INSERT, выполняются за время, пропорциональное средней длине пути в этом дереве. Затем мы показали, что средняя глубина узлов составляет $O(\log n)$ для "случайного" дерева из n узлов. Однако некоторые последовательности операций вставки и удаления узлов дерева могут привести к деревьям двоичного поиска, чья средняя глубина будет пропорциональна n. Это наводит на мысль попытаться после каждой вставки или удаления реорганизовать дерево так, чтобы оно всегда было полно, тогда время выполнения оператора INSERT и других подобных операторов всегда будет иметь порядок $O(\log n)$.

На рис. 5.5,а показано дерево из шести узлов, а на рис. 5.5,б — то же дерево (уже полное) после вставки седьмого узла с элементом 1. Но обратите внимание, что каждый узел на рис. 5.5,б имеет другого родителя, отличного от родителя, изображенного на рис. 5.5,а. Отсюда следует, что надо сделать л шагов при вставке элемента 1, чтобы сохранить дерево по возможности сбалансированным. Это значительно хуже, чем в случае простой вставки в полное дерево двоичного поиска при реализации словарей, очередей с приоритетами и других АТД, где оператор INSERT (и другие ему подобные) требует времени порядка $O(\log n)$.

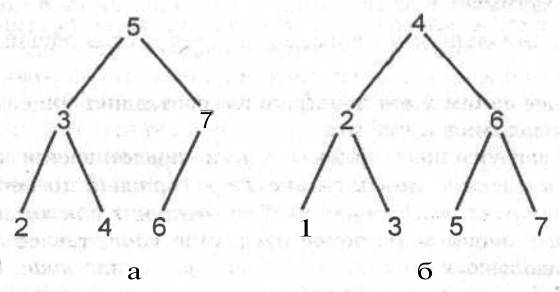


Рис. 5.5. Полные деревья

Существуют другие подходы к реализации словарей и очередей с приоритетами, где даже в самом худшем случае время выполнения операторов имеет порядок $O(\log n)$. Мы подробно рассмотрим одну из таких реализаций, которая называется *2-3 дерево* и имеет следующие свойства.

1. Каждый внутренний узел имеет или два, или три сына.
2. Все пути от корня до любого листа имеют одинаковую длину.

Будем считать, что пустое дерево и дерево с одним узлом также являются 2-3 деревьями.

Рассмотрим представления множеств, элементы которых упорядочены посредством некоторого отношения линейного порядка, обозначаемого символом "<". Элементы множества располагаются в листьях дерева, при этом, если элемент a располагается левее элемента Ъ, справедливо отношение a < Ь. Предполагаем, что упорядочивание элементов по используемому отношению линейного порядка основывается только на значениях одного поля (среди других полей записи, содержащей информацию об элементе), которое формирует тип элементов. Это поле назовем *ключом*. Например, если элементы множества — работники некоторого предприятия, то ключевым полем может быть поле, содержащее табельный номер или номер карточки социального страхования,

В каждый внутренний узел записываются ключ наименьшего элемента, являющегося потомком второго сына, и ключ наименьшего элемента — потомка

третьего сына, если, конечно, есть третий сын[1]. На рис. 5.6 показан пример 2-3 дерева. В этом и последующих примерах мы идентифицируем элементы с их ключевым полем, так что порядок элементов становится очевидным.

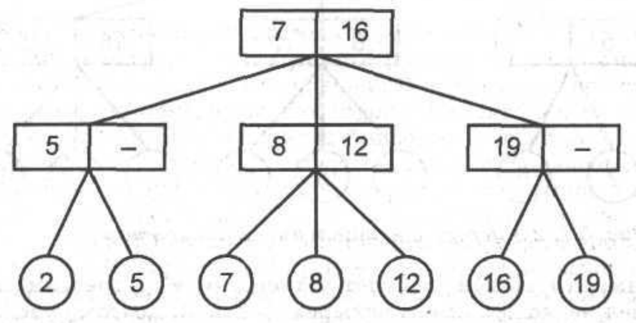


Рис. 5.6. 2-3 дерево

Отметим, что 2-3 дерево с k уровнями имеет от 2^{k-1} до 3^{k-1} листьев. Другими словами, 2-3 дерево, представляющее множество из n элементов, будет иметь не менее $1 + \log_3 n$ и не более $1 + \log_2 n$ уровней. Таким образом, длины всех путей будут иметь порядок $O(\log n)$.

Можно найти запись с ключом x в множестве, представленном 2-3 деревом, за время $O(\log n)$ путем простого перемещения по дереву, руководствуясь при этом значениями элементов, записанных во внутренних узлах. Во внутреннем узле *node* ключ x сравнивается со значением y наименьшего элемента, являющегося потомком второго сына узла *node*. (Напомним, что мы трактуем элементы так, как будто они состоят только из одного ключевого поля.) Если $x < y$, то перемещаемся к первому сыну узла *node*. Если $x > y$ и узел *node* имеет только двух сыновей, то переходим ко второму сыну узла *node*. Если узел *node* имеет трех сыновей и $x > y$, то сравниваем x со значением z — вторым значением, записанным в узле *node*, т.е. со значением наименьшего элемента, являющегося потомком третьего сына узла *node*. Если $x < z$, то переходим ко второму сыну, если $x > z$, переходим к третьему сыну узла *node*. Таким способом в конце концов мы придем к листу, и элемент x будет принадлежать исходному множеству тогда и только тогда, когда он совпадает с элементом, записанным в листе. Очевидно, что если в процессе поиска получим $x = y$ или $x = z$, поиск можно остановить. Но, конечно, алгоритм поиска должен предусмотреть спуск до самого листа.

Вставка элемента в 2-3 дерево

Процесс вставки нового элемента x в 2-3 дерево начинается так же, как и процесс поиска элемента x. Пусть мы уже находимся на уровне, предшествующем листьям, в узле *node*, чьи сыновья (уже проверено) не содержат элемента x. Если узел *node* имеет двух сыновей, то мы просто делаем элемент x третьим сыном, помещая его в правильном порядке среди других сыновей. Осталось переписать два числа в узле *node* в соответствии с новой ситуацией.

Например, если мы вставляем элемент 18 в дерево, показанное на рис. 5.6, то узлом *node* здесь будет самый правый узел среднего уровня. Помещаем элемент 18 среди сыновей этого узла, упорядочивая их как 16, 18 и 19. В узле *node* записываются числа 18 и 19, соответствующие второму и третьему сыновьям. Результат вставки показан на рис. 5.7.

---

[1] Существуют другие разновидности 2-3 деревьев, когда во внутренний узел помещаются целые записи, как это делается в деревьях двоичного поиска.

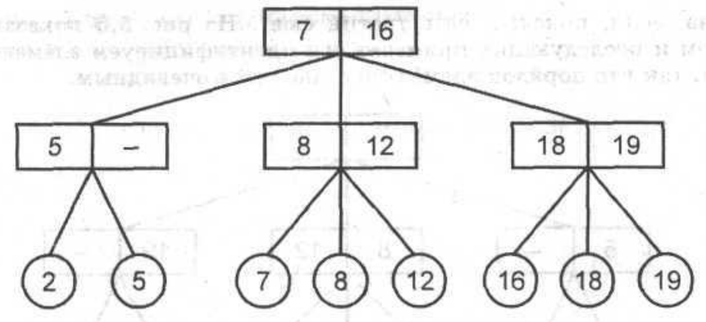


Рис. 5.7. 2-3 дерево с вставленным элементом 18

Предположим, что элемент x будет четвертым, а не третьим сыном узла *node*. В 2-3 дереве узел не может иметь четырех сыновей, поэтому узел *node* разбивается на два узла, которые назовем *node* и ***node'***. Два наименьших элемента из четырех станут сыновьями нового узла *node,* а два наибольших элемента — сыновьями узла ***node'***. Теперь надо вставить узел ***node'*** среди сыновей узла p — родителя узла *node*. Здесь ситуация аналогична вставке листа к узлу *node.* Так, если узел p имеет двух сыновей, то узел ***node'*** становится третьим (по счету) сыном этого узла и помещается непосредственно справа от узла *node.* Если узел p уже имеет трех сыновей, то он разбивается на два узла p и p', узлу p приписываются два наименьших сына, а узлу p' — оставшиеся. Затем рекурсивно вставляем узел p' среди сыновей родителя узла p и т.д.

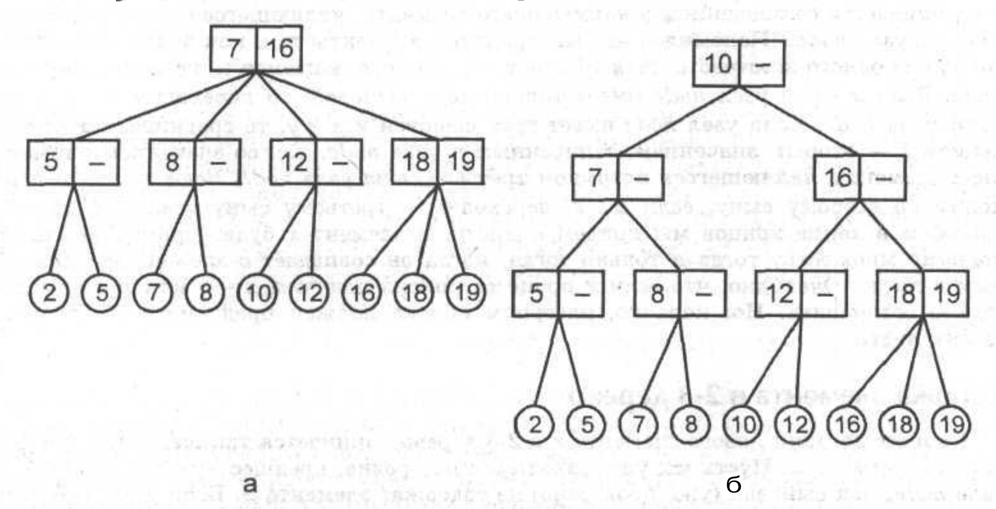


Рис. 5.8. Вставка элемента 10 в 2-3 дерево из рис. 5.7

Особый случай возникает при необходимости разбиения корня дерева. В этой ситуации создается новый корень, чьими сыновьями будут два узла, полученные в результате разбиения старого корня. Очевидно, в этом случае количество уровней 2-3 дерева возрастает на единицу.

Пример 5.4. Предположим, что надо вставить элемент 10 в дерево, показанное на рис. 5.7. Намеченный родитель для нового элемента уже имеет трех сыновей 7, 8 и 12, поэтому он разбивается на два узла. Первый узел будет иметь сыновей 7 и 8, а второй — 10 и 12. Результат этого разбиения показан на рис. **5.8,а.** Теперь необходимо вставить новый узел (с сыновьями 10 и 12) среди сыновей корня дерева. С этим

узлом корень будет иметь четыре сына, поэтому он разбивается на два узла и образуется новый корень, как показано на рис. **5.8,б.** Подробности реорганизации информации об элементах будут показаны далее при разработке программы для оператора INSERT. □

Удаление элемента из 2-3 дерева

При удалении листа может случиться так, что родитель *node* этого листа останется только с одним сыном. Если узел *node* является корнем, то единственный сын становится новым корнем дерева. Для случая, когда узел *node* не является корнем, обозначим его родителя как *р*. Если этот родитель имеет другого сына, расположённого слева или справа от узла *node,* и этот сын имеет трех сыновей, то один из них становится сыном узла *node.* Таким образом, узел *node* будет иметь двух сыновей.

Если сын родителя *р,* смежный с узлом *node,* имеет только двух сыновей, то единственный сын узла *node* присоединяется к этим сыновьям, а узел *node* удаляется. Если после этого родитель *р* будет иметь только одного сына, то повторяется описанный процесс с заменой узла *node* на узел *р.*

Пример 5.5. Удалим элемент 10 из дерева, представленного на рис. **5.8,б.** После удаления этого элемента его родитель будет иметь только одного сына. Но его "дедушка" имеет другого сына, у которого, в свою очередь, есть три сына: элементы 16, 18 и 19. Этот узел находится справа от неполного узла, поэтому лист с наименьшим элементом 16 переносится в неполный узел. В результате получим дерево, показанное на рис. **5.9,а.**

Пусть далее надо удалить из полученного дерева элемент 7. После удаления его родитель будет иметь только одного сына 8, а его "дедушка" не имеет сыновей с тремя "детьми". Поэтому делаем элемент 8 "братом" элементов 2 и 5, как показано на рис. **5.9,б.** На этом рисунке узел, помеченный звездочкой, имеет только одного сына, а его родитель не имеет сыновей с тремя "детьми". Поэтому мы удаляем помеченный узел, присоединяя его сына к узлу, расположенному справа от помеченного узла. Теперь корень будет иметь только одного сына, поэтому он также удаляется. В результате получим дерево, показанное на рис. **5.9,в.**

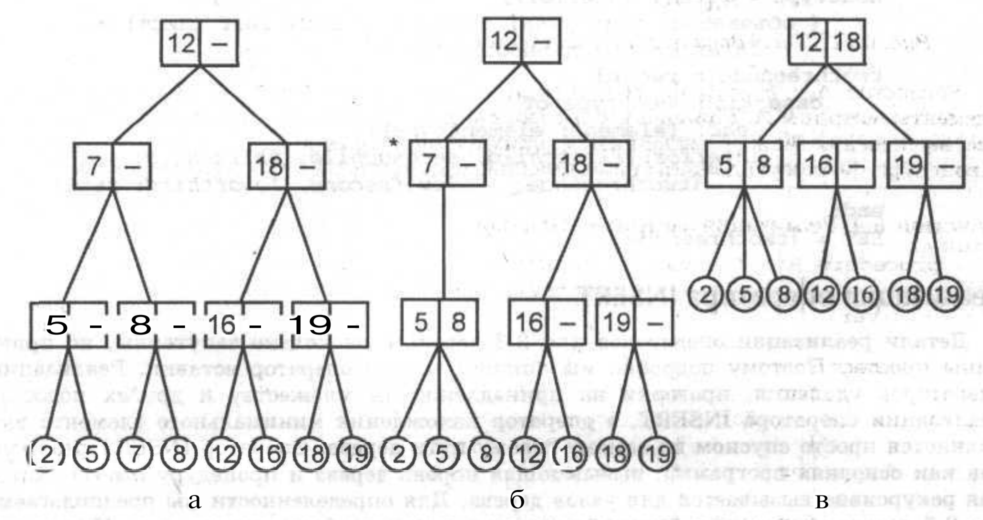


Рис. 5.9. Удаление элемента 7 из 2-3 дерева

Из вышеприведенных примеров видно, что часто приходится манипулировать значениями внутренних узлов. Мы всегда сможем вычислить эти **значения,** если будем помнить наименьшие значения всех потомков узлов, составляющих путь от де-

рева до удаляемого листа. Эту информацию можно получить путем рекурсивного применения алгоритма удаления, вызываемого для каждого узла, составляющего путь, который начинается от корня дерева. Возникающие при этом ситуации учтены при написании программы (см. далее), реализующей оператор **DELETE**. □

Типы данных для 2-3 деревьев

Ограничимся **представлением** (посредством 2-3 деревьев) множеств элементов, чьи ключи являются действительными числами. Природа других полей, которые вместе с полем ***key*** (ключ) составляют запись об элементе, пока не определена и здесь нас не интересует; общий тип записи обозначим как elementtype.

При написании программ на языке Pascal родители листьев должны быть записями, содержащими два поля действительных чисел (ключи наименьших элементов во втором и в третьем поддеревьях) и три поля указателей **на** элементы. Родители этих узлов являются записями, состоящими из двух полей действительных чисел и трех полей указателей на родителей листьев. Таким образом, разные уровни 2-3 дерева имеют разные типы данных (указатели на элементы или указатели на узлы). Эта ситуация вызывает затруднения при программировании на языке Pascal операторов, выполняемых над 2-3 деревьями. К счастью, язык Pascal имеет механизм, вариант структуры record (запись), который позволяет обращаться с узлами 2-3 дерева, имеющими разный тип данных[1]. В листинге 5.8 приведено объявление типов данных для узлов 2-3 дерева, а также для типа данных SET (Множество), представляющего 2-3 дерево, в виде указателя на корень этого дерева.

Листинг 5.8. Объявление типов данных узлов 2-3 дерева

```
type
    elementtype = record
        key: real;
        { объявление других полей, если необходимо }
    end;
    nodetype = (leaf, interior);
        { объявление типа узла, содержащее поля leaf (лист) и
            interior (внутренний узел) }
    twothreenode = record
        case kind: nodetype of
            leaf: (element: elementtype) ;
            interior:(firstchild, secondchild, thirdchild:
                ↑twothreenode;    lowofsecond, lowofthird: real)
    end;
    SET = ↑twothreenode;
```

Реализация оператора INSERT

Детали реализации операторов для 2-3 деревьев несколько запутанны, но принципы просты. Поэтому подробно мы опишем только оператор вставки. Реализации операторов удаления, проверки на принадлежность множеству и других подобны реализации оператора INSERT, а оператор нахождения минимального элемента выполняется просто спуском по самому левому пути дерева. Оператор INSERT реализован как основная программа, вызывающая корень дерева и процедуру ***insert*1**, которая рекурсивно вызывается для узлов дерева. Для определенности мы предполагаем, что 2-3 дерево не является пустым деревом и содержит более одного узла. Последние

---

[1] В данном случае все равно каждый узел будет занимать пространство, равное пространству, занимаемому самым "большим" типом **данных.** Поэтому язык Pascal — не лучший язык для практической реализации 2-3 деревьев.

два случая, когда дерево пустое или содержит только один узел, реализуются простой последовательностью шагов, которые читатель может написать самостоятельно.

Функция *insert*1 должна возвращать как указатель на новый узел, если он будет создан, так и ключ наименьшего элемента, находящегося ниже нового узла. В языке Pascal механизм создания такой функции достаточно сложен, поэтому мы объявили *insert*1 как процедуру, которая присваивает значения параметрам *pnew* и *low* при создании нового узла. В листинге 5.9 показан эскиз этой процедуры, а ее полный код приведен в листинге 5.10.

Листинг 5.9. Процедура вставки в 2-3 дерево

```
procedure insert1 ( node: ↑twothreenode; x: elementtype;
        { элемент x должен быть вставлен в поддерево с корнем node }
    var pnew: ↑twothreenode; { указатель на новый элемент }
    var low: real); { наименьший элемент в поддереве с корнем,
                                на который указывает pnew }
    begin
        pnew:= nil;
        if node является листом then begin
            if x не является элементом, содержащимся в node then begin
                создание нового узла, указанного pnew;
                сделать x элементом, содержащимся в новом узле;
                low:= x.key
            end
        end
        else begin { node является внутренним узлом }
            выбрать w — сына узла node;
            insert1(w, x, pback, lowback);
            if pback <> nil then begin
                вставка указателя pback среди детей узла node,
                    расположенных справа от w,
                if node имеет 4-х сыновей then begin
                    создание нового узла, указанного pnew;
                    создание нового узла для 3-го и 4-го сыновей node;
                    задание значений полям lowofsecond и lowofthird
                        для узла node и нового узла;
                    задание полю low наименьшего ключа
                        среди детей нового узла
                end
            end
        end
    end; { insert1 }
```

Листинг 5.10. Полный код процедуры *insert*1

```
procedure insert1 ( node: ttwothreenode;     x: elementtype;
    var pnew: Ttwothreenode; var low: real );
    var
        pback: ttwothreenode;
        lowback: real;
        child: 1..3; { указывает на следующего "обрабатываемого"
                            сына node (ср. с w в листинге 5.9) }
        w: ttwothreenode; { указатель на сына }
    begin
        pnew:= nil;
```

```
if node↑.kind = leaf then begin
    if node↑.element.key <> x.key then begin
        { создание нового листа, содержащего x и
        "возврат" этого листа }
        new(pnew, leaf);
        if node↑.element.key < x.key then
            { запись x в новый узел }
            begin pnew↑.element:= x; low:= x.key end
        else begin
            pnew↑.element:= node↑.element;
            node↑.element:= x;
            low:= pnew↑.element.key
        end
    end
end
else begin { node является внутренним узлом }
    { выбор следующего сына node }
    if x.key < node↑.lowofsecond then
        begin child:= 1; w:= node↑.firstchild end
    else if(node↑.thirdchild = nil) or
            (x.key < node↑.lowofsecond) then begin
        { x во втором поддереве }
        child:= 2;
        w:= node↑.secondchild
    end
    else begin { x в третьем поддереве }
        child:= 3;
        w:= node↑.thirdchild
    end;
    insert1(w, x, pback, lowback);
    if pback <> nil then
        { надо вставить нового сына node }
        if node↑.thirdchild = nil then
            { node имеет только двух сыновей }
            if child - 2 then begin
                node↑.thirdchild:= pback;
                node↑.lowofthird:= lowback
            end
            else begin { child = 1 }
                node↑.thirdchild:= node↑.secondchild;
                node↑.lowofthird:= node↑.lowofsecond;
                node↑.secondchild:= pback;
                node↑.lowofsecond:= lowback
            end
        else begin { node уже имеет трех сыновей }
            new(pnew, interior);
            if child = 3 then begin
            {pback и 3-й сын становятся сыновьями нового узла}
                pnew↑.firstchild:=nodeT.thirdchild;
                pnew↑.secondchild:= pback;
                pnew↑.thirdchild:= nil;
                pnew↑.lowofsecond:= lowback;
                    { lowofthird для pnew неопределен }
                low:= node↑.lowofthird;
```

```
                    node↑.thirdchild:= nil
                end
                else begin
                    {child < 2; перемещение 3-го сына node к pnew}
                    pnew↑.secondchild:= node↑.thirdchild;
                    pnew↑.lowofsecond:= node↑.lowofthird;
                    pnew↑.thirdchild:= nil;
                    node↑.thirdchild:= nil
                end;
                if child = 2 then begin
                    (pback становится 1-м сыном pnew }
                    pnew↑.firstchild:= pback;
                    low:= lowback
                end;
                if child = 1 then begin
                    { 2-й сын node перемещается к pnew,
                        pback становится 2-м сыном node }
                    pnew↑.firstchild:= node↑.secondchild;
                    low:= nodet - lowofsecond;
                    node↑.secondchild:= pback;
                    node↑.lowofsecond= lowback
                end
            end
        end
    end; { insert1 }
```

Теперь можно написать процедуру INSERT, которая вызывает *insert*1. Если *insertl* "возвращает" новый узел, тогда INSERT должна создать новый корень дерева. Код процедуры INSERT представлен в листинге 5.11. Здесь мы считаем, что тип SET — это тип Ttwothreenode, т.е. указатель на корень 2-3 дерева, чьи листья содержат элементы исходного множества.

Листинг 5.11. Оператор INSERT для множеств, представленных посредством 2-3деревьев

```
procedure INSERT ( x: elementtype; var S: SET ) ;
    var
        pback: ↑twothreenode; {указатель на узел, возвращенный insert1}
        lowback: real; {наименьшее (low) значение в поддереве pback}
        saveS: SET; { для временного хранения копии указателя S }
    begin
        { здесь надо вставить процедуру проверки, которая выясняет,
          . является ли S пустым деревом или имеет только один узел,
            и Осуществляет для этих ситуаций вставку }
        insert1(S, x, pback, lowback);
        if pback <> nil then begin
        { создание нового корня, на его сыновей указывают S и pback }
            saveS:= S;
            new(S);
            S↑.firstchild:= saveS;
            S↑.secondchild:= pback;
            S↑.lowofsecond:= lowback;
            S↑.thirdchild:= nil;
        end
    end; { INSERT )
```

Реализация оператора DELETE

Сделаем набросок функции *delete1*, которая по заданному указателю на узел *node* и элементу *x* удаляет лист, являющийся потомком узла *node* и содержащий значение *x*, если, конечно, такой лист существует[1]. Если после удаления узел *node* имеет только одного сына, то функция *delete1* возвращает значение true, а если узел *node* имеет двух или трех сыновей, то возвращает значение false. Эскиз этой функции показан в следующем листинге.

Листинг 5.12. Рекурсивная процедура удаления

```
function delete1 ( node: ↑twothreenode; x: elementtype ): boolean;
    var
        onlyone: boolean;
                { содержит значение, возвращаемое по вызову delete1 }
    begin
        delete1:= false;
        if сыновья node являются листьями then begin
            if x принадлежит этим листьям then begin
                удаление x;
                смещение сыновей node, которые были справа от x,
                        на одну позицию влево;
                if node имеет только одного сына then
                    delete1:= true
            end
        end
        else begin { node находится на втором уровне или выше }
            определяется, какой сын узла node может
                иметь среди своих потомков x;
            onlyone:= delete1(w, x); { в зависимости от ситуации w
                обозначает node↑.firstchild, node↑.secondchild или
                node↑.thirdchild }
            if onlyone then begin { просмотр сыновей node }
                if w — первый сын node then
                    if y — 2-й сын node, имеющий 3-х сыновей then
                        1-й сын y делается 2-м сыном w;
                    else begin { y имеет двух сыновей }
                        сын w делается 1-м сыном y;
                        удаление w из множества сыновей узла node;
                        if node имеет только одного сына then
                            delete1:= true
                    end;
                if w — второй сын node then
                    if y — 1-й сын node, имеющий 3-х сыновей then
                        3-й сын y делается 1-м сыном w
                    else { y имеет двух сыновей }
                        if существует z — 3-й сын node,
                            имеющий 3-х сыновей then
                                1-й сын z делается 2-м сыном w
                        else begin
                            { y node нет сыновей, имеющих трех детей }
                            сын w делается 3-м сыном y;
```

[1] Полезным вариантом такой функции была бы функция, которая только по ключевому значению удаляла все элементы с этим ключом.

```
                         удаление w из множества сыновей узла node;
                         if node имеет только одного сына then
                              delete1:= true
                    end;
               if w — третий сын node then
                    if y — 2-й сын node, имеющий 3-х сыновей then
                         3-й сын y делается 1-м сыном w;
                    else begin { y имеет двух сыновей }
                         сын w делается 3-м сыном y;
                         удаление w из множества сыновей узла node;
                    end
               end
          end
     end; { delete1 }
```

Мы оставляем для читателя детализацию кода функции *delete1*. В качестве еще одного упражнения предлагаем написать процедуру **DELETE(*S*, *x*)**, которая проверяла бы, не является ли множество S пустым или состоящим из одного элемента, и если это не так, то вызывала бы функцию ***delete*1(*S*, *x*)**. Если *delete1* возвратит значение true, то процедура должна удалить корень (узел, на который указывает S) и сделать S указателем на единственного (в данной ситуации) сына корня.

5.5. Множества с операторами MERGE и FIND

В этом разделе мы рассмотрим ситуацию, когда есть совокупность объектов, каждый из которых является множеством. Основные действия, выполняемые над такой совокупностью, заключаются в объединении множеств в определенном порядке, а также в проверке принадлежности определенного объекта конкретному множеству. Эти задачи решаются с помощью операторов MERGE (Слить) и FIND (Найти). Оператор **MERGE(*A*, *B*, *C*)** делает множество *C* равным объединению множеств *A* и *B*, если эти множества не пересекаются (т.е. не имеют общих элементов); этот оператор не определен, если множества *A* и *B* пересекаются. Функция **FIND(*x*)** возвращает множество, которому принадлежит элемент *ж;* в случае, когда *ж* принадлежит нескольким множествам или не принадлежит ни одному, значение этой функции не определено.

Пример 5.6. *Отношением эквивалентности* является отношение со свойствами рефлексивности, симметричности и транзитивности. Другими словами, если на множестве S определено отношение эквивалентности, обозначаемое символом "≡", то для любых элементов *a*, *b* и *c* из множества S (не обязательно различных) справедливы следующие соотношения.

1. a ≡ a (свойство рефлексивности).
2. Если *a* ≡ *b*, то *b* ≡ a (свойство симметричности).
3. Если *a* ≡ *b* и *b* ≡ *c*, *to a* ≡ *c* (свойство транзитивности).

Отношение равенства (обозначается знаком "=") — это пример отношения эквивалентности на любом множестве *S*. Для любых элементов *a*, *b* и c из множества S имеем (1) *a — a;* (2) если a = *b*, то *b* = *a;* (3) если a = *b* и *b* = c, то a = c. Далее мы встретимся с другими примерами отношения эквивалентности.

В общем случае отношение эквивалентности позволяет разбить совокупность объектов на непересекающиеся группы, когда элементы *a* и *b* будут принадлежать одной группе тогда и только тогда, когда *a* ≡ *b*. Если применить отношение равенства (частный случай отношения эквивалентности), то получим группы, состоящие из одного элемента.

Более формально можно сказать так: если на множестве S определено отношение эквивалентности, то в соответствии с этим отношением множество S можно разбить на непересекающиеся подмножества S_1, S_2,..., которые называются *классами эквивалентности,* и объединение этих классов совпадает с S. Таким образом, *a* = *Ъ* для всех а и & из подмножества S_i, но а не эквивалентно *Ь,* если они принадлежат разным подмножествам. Например, отношение сравнения по модулю *n*[1] — это отношение эквивалентности на множестве целых чисел. Чтобы показать это, достаточно заметить, что *а - а* = 0 (рефлексивность), если *а - Ъ* = *dn,* то *b - a* = *(-d)n* (симметричность), если *a - b* = *dn* и *b - c* = *en,* то а - *с* = *(d* + *e)n* (транзитивность). В случае сравнения по модулю *п* существует га классов эквивалентности, каждое из которых является множеством целых чисел. Первое множество состоит из целых чисел, которые при делении на *п* дают остаток 0, второе множество состоит из целых чисел, которые при делении на га дают остаток 1, и т.д., га-е множество состоит из целых чисел, которые дают остаток п — 1.

Задачу эквивалентности можно сформулировать следующим образом. Есть множество S и последовательность утверждений вида "о эквивалентно *b*". Надо по представленной последовательности таких утверждений определить, какому классу эквивалентности принадлежит предъявленный элемент. Например, есть множество *S* — {1, 2, ..., 7} и последовательность утверждений

1 = 2 **5 ≡ 6 3 ≡ 4 1 ≡ 4**

Необходимо построить **классы эквивалентности** множества S. Сначала полагаем, что все элементы этого множества представляют отдельные классы, затем, применяя заданную последовательность утверждений, объединяем "индивидуальные" классы. Вот последовательность этих объединений:

1 = 2	{1, 2} {3} {4} {5} {6} {7}
5 = 6	{1, 2} {3} {4} {5, 6} {7}
3 ≡ 4	{1, 2} {3, 4} {5, 6} {7}
1 ≡ 4	{1, 2, 3, 4} {5, 6} {7}

Можно "решать" задачу эквивалентности начиная с любого утверждения заданной последовательности утверждений. При "обработке" утверждения а = *Ь* сначала с помощью оператора FIND находятся классы эквивалентности для элементов а и ft, затем к этим классам применяется оператор MERGE.

Задача эквивалентности часто встречается во многих областях компьютерных наук. Например, она возникает при обработке компилятором Fortran "эквивалентных объявлений", таких как

EQUIVALENCE **(A(1),B(1,2),C(3)),** (A(2),D,E), **(F,G)**

Другой пример представлен в главе 7, где решение задачи эквивалентности помогает найти **остовное** дерево минимальной стоимости. П

Простая реализация АТД MFSET

Начнем с простейшего АТД, реализующего операторы MERGE и FIND. Этот АТД, назовем его MFSET (Множество с операторами MERGE и FIND), можно определить как множество, состоящее из **подмножеств-*компонентов*,** со следующими операторами.

1. **MERGE(*A*,** *B)* объединяет компоненты *А* **и** *В,* результат присваивается или *А,* или *В.*
2. **FIND(*x*)** — функция, возвращающая имя компонента, которому принадлежит *х.*
3. **INITIAL(*A*, *x*)** создает компонент с именем *А,* содержащим только элемент *х.*

[1] Говорят, что *а сравнимо с Ь по модулю п,* если а и Ь имеют **один** и тот же остаток от деления на *п,* или, другими словами, если *а — Ь* кратно га.

Для корректной реализации АТД MFSET надо разделить исходные типы данных или объявить, что MFSET состоит из данных двух разных типов — типа имен множеств и типа элементов этих множеств. Во многих приложениях можно использовать целые числа для имен множеств. Если общее количество элементов всех компонент равно л, то можно использовать целые числа из интервала от 1 до *n* для идентификации элементов множеств. Если принять это предположение, то в таком случае существенно, что номерами элементов можно индексировать ячейки массива. Тип имен компонентов не так важен, поскольку это тип ячеек массива, а не индексов. Но если мы хотим, чтобы тип элементов множеств был отличным от числового, то необходимо применить отображение, например посредством хеш-функции, ставящее в соответствие элементам множеств уникальные целые числа из заданного интервала. В последнем случае надо знать только общее число элементов всех компонентов.

После всего сказанного не должно вызывать возражений следующее объявление типов:

```
const
    л = { количество всех элементов };
type
    MFSET = array[1..л] of integer;
```

или объявление более общего типа

```
    array[интервал элементов] of (тип имен множеств)
```

Предположим, что мы объявили тип MFSET как массив *components* (компоненты), предполагая, что *components*[*x*] содержит имя множества, которому принадлежит элемент *x*. При этих предположениях операторы АТД MFSET реализуются легко, что видно из листинга 5.13 с кодом процедуры MERGE. Процедура **INITIAL(*A*, *x*)** просто присваивает *components*[*x*] значение *A*, *а* функция **FIND(*x*)** возвращает значение *components*[*x*].

Листинг 5.13. Процедура MERGE

```
procedure MERGE ( A, B: integer; var C: MFSET ) ;
    var
        x: 1..n;
    begin
        for x:= 1 to л do
            if C[x] = B then
                C[x]:= Л
    end; { MERGE }
```

Время выполнения операторов при такой реализации MFSET легко проанализировать. Время выполнения процедуры MERGE имеет порядок $O(n)$, а для INITIAL и FIND время выполнения фиксированно, т.е. не зависит от п.

Быстрая реализация АТД MFSET

Применение алгоритма из листинга 5.13 для последовательного выполнения *n* - 1 раз оператора MERGE займет время порядка $O(n^2)$[1]. Один из способов ускорения выполнения оператора MERGE заключается в связывании всех элементов компонента в отдельные списки компонентов. Тогда при слиянии компонента *B* в компонент А вместо просмотра всех элементов необходимо будет просмотреть только список элементов компонента *B*. Такое расположение элементов сократит среднее время слияния компонентов. Но нельзя исключить случая, когда каждое i-е слияние выполня-

[1] Для слияния *n* элементов в одно множество потребуется в самом худшем случае не более *n* - 1 выполнений оператора MERGE.

ется в виде оператора **MERGE(*A, B*)**, где компонент *A* состоит из одного элемента, а *B* — из *i* элементов, и результат слияния получает имя компонента *A*. Выполнение такого оператора требует *O(i)* шагов, а последовательность *n* - 1 таких операторов потребует времени порядка $\sum_{i=1}^{n-1} i = n(n-1)/2$.

Чтобы избежать описанной ситуации, можно отслеживать размер каждого компонента и всегда сливать меньшее множество в **большее**[1]. Тогда каждый раз элемент из меньшего множества переходит в множество, которое, по крайней мере, в два раза больше его исходного **множества**[2]. Поэтому, если первоначально было *n* компонентов и каждый из них содержал по одному элементу, то любой элемент может перейти в любой компонент не более чем за 1 + **log*n*** шагов. Таким образом, в новой версии оператора MERGE время его выполнения пропорционально количеству элементов в компоненте, чье имя изменяется, а общее число таких изменений не больше n(l + logn), поэтому время всех слияний имеет порядок *O(n* logn).

Теперь рассмотрим структуру данных, необходимую для такой реализации. Во-первых, необходимо отображение имен множеств в записи, состоящие из

- поля *count* (счет), содержащего число элементов в множестве, и
- поля *firstelement* (первый элемент), содержащего индекс Ячейки массива с первым элементом этого множества.

Во-вторых, необходим еще один массив записей, индексированный элементами. Здесь записи имеют поля:

- ***setname*** (имя множества), содержащее имя множества;
- ***nextelement*** (следующий элемент), значение которого **указывает** на следующий элемент в списке данного множества.

Будем использовать 0 в качестве маркера конца списка, т.е. значения NIL. В языках программирования, подходящих для таких структур, можно было бы использовать указатели на последний массив, но язык Pascal не разрешает указателей внутрь массивов.

В специальном случае, когда имена множеств и элементы являются целыми числами из интервала от **1** до п, можно использовать массив для реализации отображения, описанного выше. В таком случае возможно следующее объявление:

```
type
    nametype = 1..n;
    elementtype = 1..n;
    MFSET =' record
        setheaders: array[1..n] of record
            { заголовки списков множеств }
            count: 0..n;
            firstelement: 0..n;
        end;
        names: array[1..n] of record
            { массив имен множеств }
            setname: nametype;
            nextelement: 0..n
        end
    end;
```

[1] Отметим здесь полезность и важность присваивания результата слияния наименьшему множеству, хотя в более простых реализациях результат слияния обычно присваивается первому аргументу оператора MERGE.

[2] Здесь **"намекается"** на то, что если последовательность применений оператора MERGE представить в виде дерева, то это дерево по своей структуре будет приближаться к полному дереву. Отсюда вытекает утверждение следующего предложения о "1 + **logn** шагах". — *Прим. ред.*

Процедуры INITIAL, MERGE и FIND представлены в листинге 5.14. На рис. 5.10 показан пример описанной структуры данных, состоящей из множеств 1 = {1, 3, 4}, 2 - {2} и 5 = {5, 6}.

Листинг 5.14. Операторы АТД MFSET

```
procedure INITIAL ( A: nametype; x: elementtype; var C: MFSET );
    { инициализирует множество с именем A, содержащее элемент x }
    begin
        C.names[x].setname:= A;
        C.names[x].nextelement:= 0;
            { нулевой указатель на конец списка элементов A }
        C.setheaders[A].count:= 1;
        C.setheaders[A].firstelement:= x
    end; { INITIAL }

procedure MERGE ( A, B: nametype; var C: MFSET );
    { Слияние множеств A и B. Результат присваивается
        меньшему множеству }
    var
        i: O..п; {используется для нахождения конца меньшего списка}
    begin
        if C.setheaders[A].count > C.setheaders[B].count then begin
            { A большее множество, поэтому сливается B в A }
            { находится конец B, изменяется имя на A }
            i:= C.setheaders[B].firstelement;
            while C.names[i].nextelement <> 0 do begin
                C.names[i].setname:= A;
                i:= C.names[i].nextelement
            end;
            { список A добавляется в конец списка B,
                результату слияния присваивается имя A }
            { теперь i — индекс последнего элемента B }
            C.names[i].setname:= A;
            C.names[i].nextelement:= C.setheaders[A].firstelement;
            C.setheaders[A].firstelement:=
                                        C.setheaders[B].firstelement;
            C.setheaders[A].count:= C.setheaders[A].count +
                                        C.setheaders[B].count;
            C.setheaders[B].count:= 0;
            C.setheaders[B].firstelement:= 0
                { последние два шага необязательны, т.к.
                        множества B больше не существует }
        end
        else { B по крайней мере не меньше A }
            { код для этого случая такой же,
                    как и выше, с заменой местами A и B }
    end; { MERGE }

function FIND ( x: 1..n; var C: MFSET );
    { возвращает имя множества, которому принадлежит элемент x }
    begin
        return(C.names[x].setname)
    end; { FIND }
```

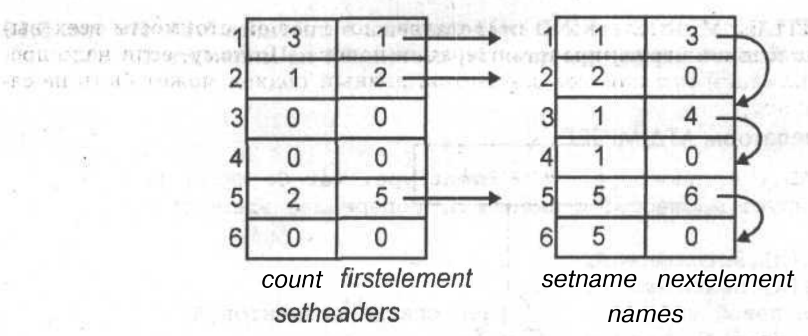


Рис. 5.10. Пример структуры данных MFSET

Реализация АТД MFSET посредством деревьев

Другой, полностью отличный о ранее рассмотренных, подход к реализации АТД MFSET применяет деревья с указателями на родителей. Мы опишем этот подход неформально. Основная идея здесь заключается в том, что узлы деревьев соответствуют элементам множеств и есть реализация, посредством массивов или какая-либо другая, отображения множества элементов в эти узлы. Каждый узел, за исключением корней деревьев, имеет указатель на своего родителя. Корни содержат как имя компонента-множества, так и элемент этого компонента. Отображение из множества имен к корням деревьев позволяет получить доступ к любому компоненту.

На рис. 5.11 показаны множества А = {1, 2, 3, 4}, В = {5, 6} и С = {7}, представленные в описанном виде. На этом рисунке прямоугольники являются частью корней, а не отдельными узлами.

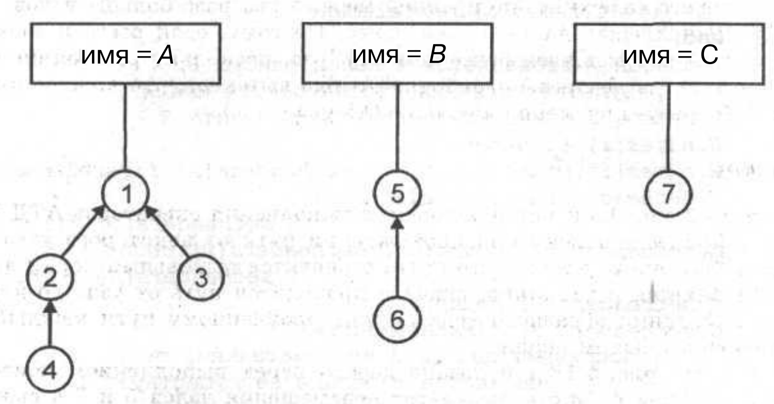


Рис. 5.11. Представление АТД MFSET набором деревьев

Чтобы определить имя компонента, которому принадлежит элемент x, мы сначала с помощью отображения (т.е. массива, который не показан на рис. 5.11) получаем указатель на узел, соответствующий элементу x. Далее следуем из этого узла к корню дерева и там читаем имя множества.

Операция слияния делает корень одного дерева сыном корня другого. Например, при слиянии множеств A и B (см. рис. 5.11), когда результат слияния получает имя А, узел 5 становится сыном узла 1. Результат слияния показан на рис. 5.12. Однако неупорядоченные слияния могут привести к результату в виде дерева из n узлов, которое будет простой цепью узлов. В этом случае выполнение оператора FIND для любого узла такого дерева потребует времени порядка $O(n^2)$. В этой связи заметим, что

хотя слияние может быть выполнено за $O(1)$ шагов, но в общей стоимости всех выполняемых операций может доминировать операция поиска. Поэтому, если надо просто выполнить *m* слияний и п поисков элементов, данный подход может быть не самым лучшим выбором.

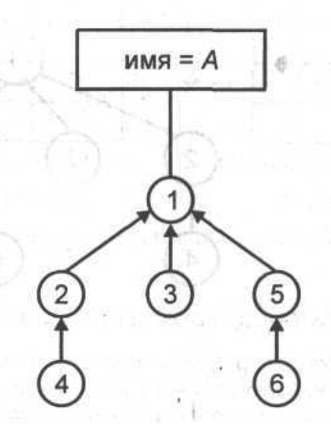


Рис. 5.12. Слияние множества A в множество B

Однако есть способ, гарантирующий при наличии *n* элементов выполнение операции поиска не более чем за $O(\log n)$ шагов. Надо просто в каждом корне хранить количество элементов данного множества, а когда потребуется слияние двух множеств, корень меньшего дерева станет сыном корня большего дерева. В этом случае при перемещении узла в новое дерево расстояние от этого узла до корня увеличится на единицу, а новое дерево содержит, по крайней мере, в два раза больше узлов, чем дерево, которому принадлежал данный узел ранее. Поэтому, если всего п элементов, то нет узла, который перемещался более чем $\log n$ раз, поскольку расстояние от любого узла до корня никогда не превышает logn. Отсюда вытекает, что каждое выполнение оператора FIND требует времени не больше $O(\log n)$.

Сжатие путей

Сжатие путей еще один метод ускорения выполнения операторов АТД MFSET. В этом методе в процессе поиска при прохождении пути от некоторого узла до корня все узлы, расположенные вдоль этого пути, становятся сыновьями корня дерева. Эта операция выполняется в два этапа: сначала проходится путь от узла до корня, а затем при осуществлении обратного хода по уже пройденному пути каждый узел поочередно становится сыном корня.

Пример 5.7. На рис. **5.13,а** показано дерево перед выполнением поиска узла с элементом 7, а на рис. **5.13,б** — результат перемещения узлов 5 и 7 в сыновья корня. Узлы 1 и 2 не перемещаются, так как узел 1 — это корень, а узел 2 уже является сыном корня. П

Сжатие путей не влияет на выполнение оператора MERGE, поскольку этот оператор выполняется за фиксированное время, но ускоряет последующие выполнения оператора FIND, так как укорачиваются пути от узлов к корню и вместе с тем на это затрачивается относительно немного усилий.

К сожалению, анализ среднего времени выполнения операторов FIND с использованием сжатия путей очень труден. Будем исходить из того, что если не требовать, чтобы меньшие деревья сливались в большие, то на выполнение *n* операторов FIND потребуется время, не превышающее $O(n \log n)$. Конечно, на выполнение первых операторов FIND может потребоваться время порядка $O(n)$ на один оператор, если деревья состоят из одной цепочки узлов. Но сжатие путей может очень быстро изменить

дерево, поэтому порядок, в котором применяется оператор **FIND** к элементам любого дерева при выполнении подряд л операторов FIND, несуществен. Но отметим, что существуют последовательности из операторов MERGE и FIND, которые требуют времени порядка $\Omega(n \log n)$.

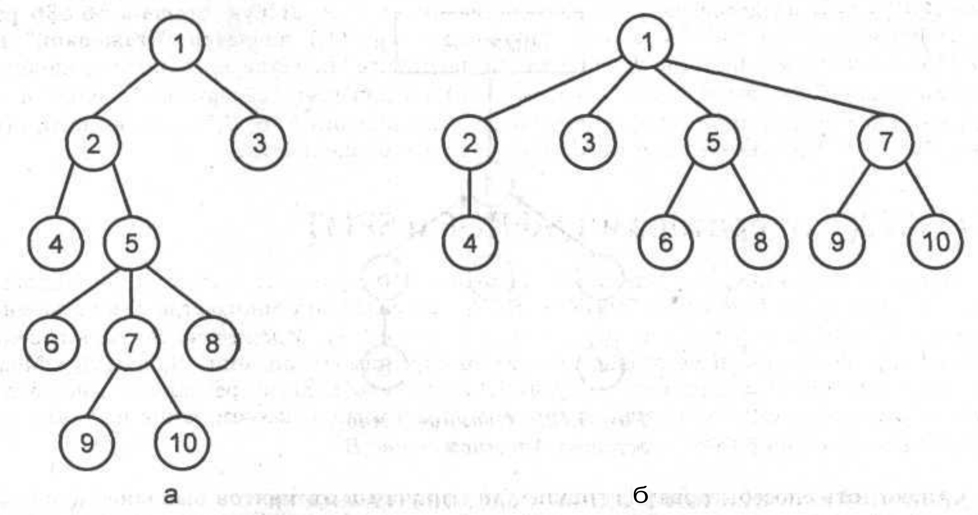


Рис. 5.13. Пример сжатия пути

Алгоритм, использующий как сжатие путей, так и слияние меньших множеств в большие, — асимптотически наиболее эффективный метод (из известных) реализации АТД MFSET. На практике л выполнений оператора поиска требует времени, не превышающего $O(n\ \alpha(n))$, где а(л) — функция, которая при возрастании л растет значительно медленнее, чем $\log n$. Мы определим функцию $\alpha(n)$ ниже, но **анализ,** который приводит к такой **оценке,** выходит за рамки этой книги.

Функция $\alpha(n)$

Функция а(л) тесно связана с очень быстро растущей функцией $A(x, y)$, известной как *функция* ***Аккермана.*** Эта функция рекуррентно определяется следующими соотношениями:

$A(0, y) = 1$ для любого $y > 0$,
$A(1, 0) = 2$,
$A(x, 0) = x + 2$ для любого $x > 2$,
$A(x, y) = A(A(x - 1, y), y - 1)$ для всех $x, y > 1$.

Фиксированное значение y определяет функцию одной переменной. Например, третье вышеприведенное соотношение для $y = 0$ определяет функцию "прибавить 2". Для $y = 1$ и $x > 1$ имеем $A(x, 1) = A(A(x - 1, 1), 0) = A(x - 1, 1) + 2$ с начальным значением $A(1, 1) = A(A(0, 1), 0) = A(1, 0) = 2$. Отсюда следует, что $A(x, 1) = 2x$ для всех $x > 1$. Другими словами, $A(x, 1)$ — это "умножение на 2". Аналогично, для $y = 2$ и $x > 1$ получим $A(x, 2) = A(A(x - 1, 2), 1) = 2A(x - 1, 2)$ и $A(1, 2) = A(A(0, 2), 1) = A(1, 1) = 2$. Таким образом, $A(x, 2) = 2^x$. Подобным образом можно показать, что $A(x, 3) = 2^{2^{\cdot^{\cdot^{2}}}}$ ("этажерка" из x двоек). В свою очередь $A(x, 4)$ растет так быстро, что этот рост нельзя показать с помощью обычной математической записи.

Функцию Аккермана одной переменной можно также определить **как** $A(x) — A(x, x)$. Тогда функция а(л) является псевдообратной к функции $A(x)$. Точнее,

$\alpha(n)$ равна наименьшему x такому, что $n < A(x)$. Например, $A(1) = 2$, поэтому $\alpha(1) = \alpha(2) = 1$. Аналогично, $A(2) = 4$, отсюда следует, что $\alpha(3) = \alpha(4) = 2$. Далее, $A(3) = 8$, поэтому $a(5) = ... = \alpha(8) = 3$.

Может показаться, что $\alpha(n)$ растет хоть и медленно, но все-таки заметно. Однако уже $A(4)$ — это последовательное возведение числа 2 во вторую степень 65 536 раз (т.е. "этажерка" из 65 536 двоек). Поскольку $\log(A(4))$ является "этажеркой" из 65 535 двоек, то мы не можем даже точно прочитать значение A(4) или определить, какова разрядность этого числа. Поэтому $\alpha(n) < 4$ для всех "обозримых" целых n. И тем не менее $\alpha(n)$, в принципе, может достигать значений 5, 6, 7, ..., более того, $\alpha(n)$ невообразимо медленно, но все-таки стремится к бесконечности.

5.6. АТД с операторами MERGE и SPLIT

Пусть S — множество, элементы которого упорядочены посредством отношения "<". Оператор разбиения SPLIT(S, S_1, S_2, x) разделяет множество S на два множества: $S_1 = \{a \mid a \in S \text{ и } a < x\}$ и $S_2 = \{a \mid a \in S \text{ и } a > x\}$. Множество S после разбиения не определено (если не оговорено, что оно принимает значение S_1 или S_2). Можно привести много различных ситуаций, когда необходимо разделить множество путем сравнения всех его элементов с одним заданным значением. Одна из таких ситуаций рассмотрена ниже.

Задача наибольшей общей подпоследовательности

Подпоследовательность последовательности x получается в результате удаления нескольких элементов (не обязательно соседних) из последовательности x. Имея две последовательности x и y, *наибольшая общая подпоследовательность* (*НОП*) определяется как самая длинная последовательность, являющаяся подпоследовательностью как x, так и y.

Например, для последовательностей 1, 2, 3, 2, 4, 1, 2 и 2, 4, 3, 1, 2, 1 НОП составляет подпоследовательность 2, 3, 2, 1, как показано на рис. 5.14. В этом примере существуют и другие НОП, в частности 2, 3, 1, 2, но нет ни одной общей подпоследовательности длиной 5.

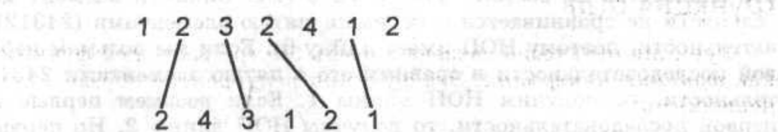


Рис. 5.14. Наибольшая общая подпоследовательность

Существует команда UNIX *diff*, которая, сравнивая файлы строка за строкой, находит наибольшую общую подпоследовательность, при этом рассматривая каждую строку файла как отдельный элемент подпоследовательности (т.е. здесь целая строка является аналогом целых чисел из примера рис. 5.14). После выполнения команды *diff* строки, не вошедшие в НОП, могут быть удалены, изменены или перемещены из одного файла в другой. Например, если оба файла являются версиями одной и той же программы, выполненными с разницей в несколько дней, то оператор *diff*, скорее всего, найдет расхождения в этих версиях.

Существуют различные способы решения задачи НОП, которые требуют выполнения порядка $O(n^2)$ шагов для последовательностей длиной n. Команда *diff* использует дифференцированную стратегию, которая работает хорошо, если файлы не имеют слишком больших повторений в строках. Например, в программах обычно много повторяющихся строк "begin" и "end", но другие строки не должны повторяться также часто.

Алгоритм, использующий команду *diff* для поиска НОП, можно применить в эффективной реализации множеств с операторами MERGE и SPLIT. В этом случае время выполнения алгоритма поиска НОП составит $O(p\log n)$, где n — максимальное число строк в файле, а p — число пар позиций с совпадающими строками, когда одна позиция из одного файла, а другая из другого. Например, для последовательностей из рис. 5.14 число p равно 12. Две единицы в каждой последовательности образуют 4 пары, три двойки в верхней последовательности и две в нижней дадут 6 пар, а тройки и четверки — еще 2 пары. В самом худшем случае p может иметь порядок га^2, тогда алгоритм поиска НОП будет выполняться за время $O(n^2 \log n)$. На практике p обычно близко к n, поэтому можно ожидать время выполнения порядка $O(n \log n)$.

Перед началом описания алгоритма предположим, что есть две строки (последовательности) элементов $A = a_1a_2...a_n$ и $B = b_1b_2...b_m$, для которых ищется НОП. Первым шагом алгоритма будет составление для каждого элемента а списка позиций строки *A,* на которых стоит этот элемент. Таким способом определяется множество PLACES(a) = $\{ i \setminus a = a_i \}$. Для вычисления множеств PLACES(a) можно создать отображение из множества элементов в заголовки списков позиций. При использовании хеш-таблиц можно вычислить множества PLACES(a) в среднем за $O(n)$ "шагов", где "шаг" — это действия, выполняемые при обработке одного элемента. Время, затрачиваемое на выполнение одного "шага", будет константой, если элемент, например, является символом или числом. Но если элементы в последовательностях *A* и *B* являются текстовыми строками, то время выполнения "шага" будет зависеть от средней длины текстовых строк.

Имея построенные множества PLACES(a) для каждого элемента а последовательности *A,* можно приступать непосредственно к нахождению НОП. Упрощая изложение материала, мы покажем только, как определить длину НОП, оставляя в качестве упражнения построение конкретных НОП. Алгоритм будет по очереди просматривать все *bj,* где *j* пробегает значения 1, 2, ..., *m.* После обработки очередного b_j мы должны для каждого i от 0 до n знать длину НОП для последовательностей $a_1 \ldots a_i$ и $b_1, ..., bj$.

Значения позиции группируются в множества S_k (k = О, 1, ..., n), состоящие из всех целых чисел (позиций) i таких, что существует НОП длины k для последовательностей $a_1, ..., a_i$ и $b_1, ..., bj$. Отметим, S_k всегда являются множествами последовательных целых чисел и для любого k числа в S_{k+1} больше, чем в S_k.

Пример 5.8. Вернемся к последовательностям рис. 5.14, и пусть j = 5. Очевидно, что множество S_0 состоит только из 0 (т.е. никакой элемент из первой последовательности не сравнивается с первыми пятью элементами (24312) из второй последовательности, поэтому НОП имеет длину 0). Если мы возьмем первый элемент из первой последовательности и сравним его с пятью элементами 24312 второй последовательности, то получим НОП длины 1. Если возьмем первые два элемента 12 из первой последовательности, то получим НОП длины 2. Но первые три элемента 123 при сравнении с элементами 24312 все равно дадут НОП длины 2. Продолжая этот процесс, получим $S_0 = \{0\}$, $S_1 = \{1\}$, $S_2 = \{2, 3\}$, $S_3 = \{4, 5, 6\}$ и $S_4 = \{7\}$. П

Предположим, что мы имеем множества S_k для (j - 1)-й позиции второй последовательности и теперь хотим изменить их для позиции *j.* Рассмотрим множество PLACES(b_j). Для каждого r из PLACES(b_j) определим, можно ли построить какую-либо НОП путем добавления совпадения a_r *с bj* к ранее построенным НОП для $a_1 \ldots a_{r-1}$ и $b_1, ..., bj$. Если оба числа r - 1 и r входят в множество S_k, тогда все числа s, s > r из S_k перейдут в множество S_{k+1} при добавлении b_j. Это следует из того, что если есть k совпадений между подпоследовательностями $a_1, ..., a_{r-1}$ и $b_1, ..., b_{j-1}$, то добавление совпадения между a_r и b_j увеличит общее число совпадений на единицу. Таким образом, для преобразования множеств S_k и S_{k+1} следует выполнить такие действия.

1. Применить оператор FIND(r) для нахождения множества S_k.
2. Если FIND(r - 1) $\neq S_k$, тогда при добавлении совпадения b_j и a_r новые НОП не образуют, множества S_k и S_{k+1} не изменяются, и надо пропустить следующие шаги.

3. Если FIND(r - 1) = S_k, то применяется оператор SPLIT(S_k, S_k, S'_k, r) для отделения от множества S_k тех чисел, которые больше или равны r.
4. Для перемещения "отделенных" элементов в множество S_{k+1} применяется оператор MERGE(S'_k, S_{k+1}, S_{k+1}).

Следует первыми рассмотреть большие позиции из множества PLACES(b_j). Чтобы увидеть, почему надо это делать, предположим, например, что множество PLACES(b_j) содержит числа 7 и 9 и до ввода в рассмотрение элемента b_j имеем S_3 = {6, 7, 8, 9} и S_4 = {10, 11}.

Если мы сначала рассмотрим позицию 7 перед позицией 9, то множество S_3 надо будет разбить на множества S_3 = {6} и S'_3 = {7, 8, 9}, затем преобразовать множество S_4: S_4 = {7, 8, 9, 10, 11}. Если затем рассмотреть позицию 9, то множество S_4 разбиваем на множества S_4 = {7, 8} и S'_4 = {9, 10, 11}, последнее множество сливается с множеством S_5. Таким образом, позиция 9 переместилась из множества S_3 в множество S_5 только при рассмотрении одного элемента из второй последовательности, что невозможно. Это означает, что в созданную НОП длины 5 включены совпадения *bj* и с a_7, и с a_9 одновременно, что является безусловной ошибкой.

В листинге 5.15 показан эскиз алгоритма построения множеств S_k при прохождении по элементам второй последовательности. Для нахождения длины НОП после выполнения этого алгоритма надо выполнить FIND(n).

Листинг 5.15. Программа нахождения НОП

```
     procedure НОП;
         begin
(1)          инициализация S0= {0, 1, . . . , п) и Sk= 0 для k от 1 до п;
(2)            for j:= 1 to n do { вычисление всех Sk для позиции j }
(3)                for наибольшее число r ∈ PLACES (bj) do begin
(4)                    k:= FIND(r) ;
(5)                    if k:= FIND(r-l) then begin
(6)                        SPLIT (Sk, Sk, S'k, r);
(7)                        MERGE (S'k, Sk+1, Sk+1)
                       end
                   end
         end; { НОП }
```

Анализ времени выполнения алгоритма нахождения НОП

Как упоминалось ранее, описываемый алгоритм целесообразно применять тогда, когда между элементами рассматриваемых последовательностей не очень много совпадений. Число совпадений можно подсчитать по формуле $p = \sum_{j=1}^{m} |\text{PLACES}(b_j)|$, где |PLACES($b_j$)| обозначает число элементов в множестве PLACES(b_j). Другими словами, p — это сумма по всем *bj* количества позиций в первой последовательности, совпадающих с b_j. Напомним, что при сравнении файлов мы ожидали, что p имеет такой порядок, как длины сравниваемых последовательностей (файлов) m и n.

Оказывается, что 2-3 деревья — подходящая структура для множеств S_k. Мы можем инициализировать эти множества (строка (1) в листинге 5.15) за $O(n)$ шагов. Для реализации функции FIND необходим массив, осуществляющий отображение из множества позиций в множество листьев дерева, а также указатели на родителей узлов в 2-3 дереве. Имя множества, т.е. k для множества S_k, можно хранить в корне дерева. В этом случае оператор FIND можно выполнить за время $O(\log n)$, следуя за указателями от листа дерева к его корню. Поэтому все выполнения строк (4) и (5) в листинге 5.15 в совокупности требуют времени порядка $O(p \log n)$.

Выполняемый в строке (5) оператор MERGE обладает тем свойством, что каждый элемент в множестве S'_k меньше, чем любой элемент в множестве S_{k+1}, и мы можем

воспользоваться этим, применяя 2-3 деревьев для реализации данного **оператора**[1]. Вначале оператор MERGE помещает 2-3 дерево множества S'_k слева от дерева множества S_{k+1}. Если оба дерева имеют одинаковую высоту, то создается новый корень, **сыновьями** которого являются корни этих MERGE деревьев. Если дерево множества S'_k короче дерева множества S_{k+1}, то корень дерева множества S'_k становится самым левым сыном самого левого узла на соответствующем уровне дерева множества S_{k+1}. Если после этого узел будет иметь четырех сыновей, то дерево модифицируется точно так же, как и при выполнении процедуры INSERT из листинга 5.11. Пример подобного объединения деревьев показан на рис. 5.15. Аналогично, если дерево множества S_{k+1} короче дерева множества *S'k*, то корень дерева множества S_{k+1} становится самым правым сыном самого правого узла на соответствующем уровне дерева множества S'_k.

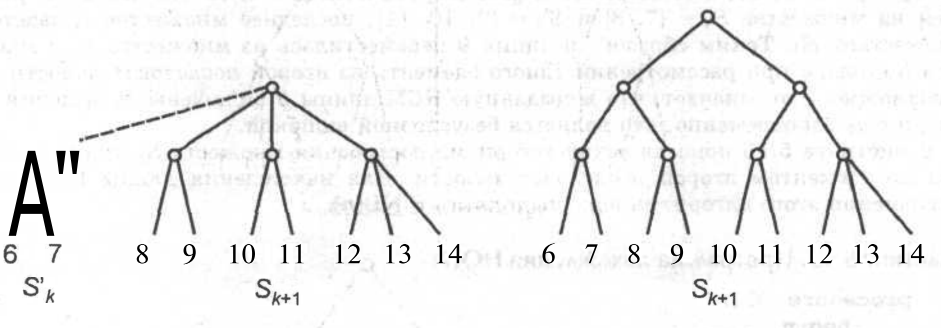


Рис. 5.15. Пример выполнения оператора MERGE

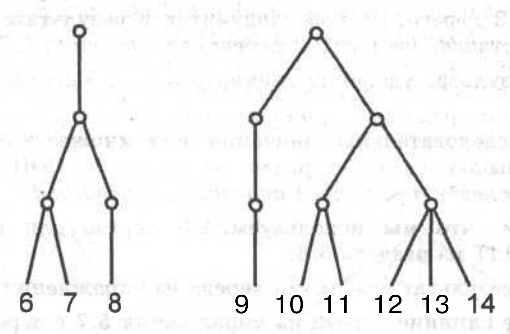
Оператор SPLIT, разбивающий множество по элементу ***r***, осуществляет проход по дереву от листа, содержащего г, к корню дерева, дублируя по пути все внутренние узлы, расположенные на пути, и отдавая эти копии узлов двум результирующим деревьям. Узлы, не имеющие сыновей, исключаются из дерева, а узлы, имеющие по одному сыну, удаляются, вставляя своего сына на соответствующий уровень дерева.

Пример 5.9. Предположим, необходимо разбить дерево, изображенное на рис. **5.15,б**, по узлу 9. Два дерева с дубликатами узлов, расположенных на пути от листа 9 к корню, показаны на рис. **5.16,а**. На дереве слева родитель листа 8 имеет только одного сына, поэтому лист 8 становится сыном родителя узлов 6 и 7. Этот родитель теперь имеет трех сыновей, поэтому все в порядке. Если бы он имел четырех сыновей, то потребовалось бы создать новый узел и вставить его в дерево. Теперь надо удалить узлы без сыновей (старый родитель листа 8) и цепочку из узлов с одним сыном, ведущую к корню дерева. После этих удалений родитель листьев 6, 7 и 8 становится корнем дерева, как показано на рис. **5.16,б**. Аналогично, на правом дереве рис. **5.16,а** лист 9 становится левым братом листьев 10 и 11, лишние узлы удаляются, в результате получаем дерево, показанное на рис. **5.16,б**. □

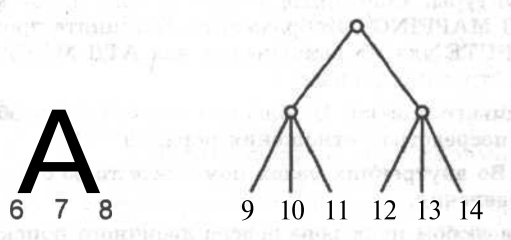
Если операция разбиения деревьев и последующая их реорганизация происходят так, как описано выше, то можно показать, что в большинстве случаев для выполнения оператора SPLIT потребуется не более $O(\log n)$ шагов. Таким образом, общее время выполнения строк (6) и (7) в листинге 5.15 требует $O(p \log n)$ шагов. Еще необходимо учесть время, затрачиваемое перед выполнением процедуры НОП на вычисление и сортировку множеств **PLACES(*a*)** для всех элементов а. Как уже упоминалось, если элементы а — "большие" объекты (например, текстовые строки), то это время может быть больше времени, необходимого на реализацию любой части описываемо-

[1] Строго говоря, мы должны дать другое **название** этому оператору, поскольку в данном случае он не объединяет непересекающиеся множества, а работает с деревьями, т.е. не соответствует названию MERGE.

го алгоритма. Как будет показано в главе 8, если манипулирование и сравнение элементов можно "уложить" в единичные "шаги", то на сортировку первой последовательности $a_1a_2...a_n$ потребуется порядка $O(n \log n)$ таких шагов, после чего PLACES(a) может быть считан из списка за время $O(n)$. В итоге получаем, что длину НОП можно подсчитать за время порядка $O(\max(n, p) \log n)$, которое (поскольку обычно $p > n$) можно принять как $O(p \log n)$.



а. Разделенные деревья



б. Результат реструктуризации

Рис. 5.16. Пример выполнения оператора SPLIT

Упражнения

5.1. Нарисуйте все возможные деревья двоичного поиска для четырех элементов 1, 2, 3 и 4.

5.2. Вставьте целые числа 7, 2, 9, 0, 5, 6, 8, 1 в пустое дерево двоичного поиска, повторив действия процедуры INSERT из листинга **5.2**.

5.3. Покажите результат удаления числа 7, а затем числа 2 из дерева, полученного в упражнении 5.2.

***5.4.** Если для удаления двух элементов из дерева двоичного поиска используется процедура из листинга 5.4, то зависит ли вид конечного дерева от порядка, в котором удаляются элементы?

5.5. Вы хотите, используя нагруженное дерево, отследить все **5-символьные** последовательности, имеющиеся в данной строке. Покажите нагруженное дерево, полученное для 14 последовательностей длины 5, извлеченных из строки ABCDABACDEBACADEBA.

5.6.** Для выполнения примера **5.5** надо в каждом листе, который представляет, например, строку ***abcde, хранить указатель на внутренний узел, представляющий (в данном случае) суффикс *bcde*. В этом случае, если, например, **следую-**

щий символ /, мы не сможем вставить всю последовательность **bcdef**, начиная процесс от корня. Также отметим, что имея последовательность **abcde**, можно создать узлы для последовательностей **bcde**, *cde*, *de* и *e*, которые в итоге будут нам необходимы. Измените структуру данных нагруженного дерева для поддержки таких указателей и модифицируйте алгоритм вставки в нагруженное дерево с учетом особенностей этой структуры данных.

5.7. Нарисуйте 2-3 дерево, которое получится в результате вставки в пустое множество (представленное как 2-3 дерево) элементов 5, 2, 7, 0, 3, 4, 6, 1, 8, 9.

5.8. Покажите результат удаления элемента 3 из 2-3 дерева, полученного в упражнении **5.7**.

5.9. Покажите последовательные значения всех множеств S_k, которые получаются в процессе выполнения алгоритма нахождения НОП из листинга 5.15 при сравнении последовательностей *abacabada* и **bdbacbad**.

5.10. Предположим, что мы используем 2-3 дерево для реализации операторов MERGE и SPLIT из раздела 5.6:

а) покажите результат разбиения дерева из упражнения 5.7 по элементу 6;

б) выполните слияние дерева из упражнения 5.7 с деревом, состоящим из листьев элементов 10 и 11.

5.11. Некоторые структуры, описанные в этой главе, легко модифицировать для реализации АТД MAPPING (Отображение). Напишите процедуры **MAKENULL**, ASSIGN и COMPUTE для их выполнения над АТД MAPPING при использовании следующих структур данных:

а) деревья двоичного поиска. В области определения отображения упорядочьте элементы посредством отношения порядка "<";

б) 2-3 деревья. Во внутренних узлах поместите только поле ключей элементов области определения.

5.12. Покажите, что в любом поддереве дерева двоичного поиска минимальный элемент находится в узле без левого сына.

5.13. Используйте пример **5.12** для создания нерекурсивной версии процедуры **DELETEMIN**.

5.14. Напишите процедуры ASSIGN, **VALUEOF**, MAKENULL и GETNEW для узлов нагруженного дерева, представленных в виде списков.

***5.15.** Сравните время выполнения операторов и используемое пространство нагруженного дерева (реализуемого как список ячеек), открытой хеш-таблицы и дерева двоичного поиска, если элементы исходного множества являются строками, состоящими не более чем из 10 символов.

***5.16.** Если элементы множества упорядочены посредством отношения "<", тогда во внутренних узлах 2-3 дерева можно хранить один или два элемента (не обязательно ключи), и в этом случае не обязательно, чтобы элементы хранились в листьях дерева. Напишите процедуры INSERT и DELETE для 2-3 дерева этого типа.

5.17. Другая модификация ,2-3 деревьев предполагает хранение во внутренних узлах дерева только ключей k_1 и k_2, но не обязательно именно минимальных ключей второго и третьего поддеревьев. Требуется только, чтобы все ключи k третьего поддерева удовлетворяли неравенству $k \geq k_2$, все ключи k второго поддерева — неравенствам $k_1 < k < k_2$, а для всех ключей k первого поддерева выполнялось $k <$ ftj.

а) как в рамках этих соглашений упростить реализацию оператора DELETE?

б) какие из операторов словарей и отображений при использовании таких 2-3 **деревьев** будут выполняться более эффективно, а какие менее?

5.18.** Еще одной структурой данных, которая поддерживает словари с оператором MIN, являются ***АВЛ-деревья (названные так в честь авторов **Г.М.** Адельсона-Вельского и **Е.М.** Ландиса), или ***деревья, сбалансированные по высоте.*** Эти деревья являются деревьями двоичного поиска, у которых для каждого узла высоты его правого и левого поддеревьев отличаются не более чем на единицу. Напишите процедуры реализации операторов INSERT и DELETE при использовании АВЛ-деревьев.

5.19. Напишите на языке Pascal программу для процедуры ***insert1*** из листинга 5.12.

5.20.** *Конечный автомат* состоит из множества состояний, которые мы будем обозначать целыми числами от 1 до *n,* таблицы ***переход[состояние, вход], которая для каждого *состояния* и для каждого символа *вход* определяет *следующее состояние.* Предположим, что входом всегда является или 0 или 1. Далее, определенные состояния назначаются ***допускающими состояниями.*** Для наших целей предположим, что только состояния, выражающиеся четными числами, являются допускающими. Два состояния называются *эквивалентными,* если являются одним и тем же состоянием, или (1) они являются допускающими или оба недопускающими; (2) вход 0 они преобразуют в эквивалентные состояния и (3) вход 1 они также преобразуют в эквивалентные состояния. Очевидно, что эквивалентные состояния "работают" одинаково на всех последовательностях входов, т.е. на основании конкретных входов оба или достигнут множества допускающих состояний, или не достигнут этого множества. Напишите программу, использующую операторы АТД **MFSET**, которая бы вычисляла множества эквивалентных состояний для данного конечного автомата.

****5.21.** Для реализации АТД MFSET посредством деревьев покажите, что

а) если используется сжатие путей и разрешается сливать только большие деревья в меньшие, то для выполнения п операций слияния требуется время порядка $\Omega(n \log n)$;

б) если используется сжатие путей, но всегда только меньшие деревья сливаются в большие, то для выполнения *n* операций слияния в самом худшем случае требуется время порядка $O(n\ \alpha(n))$.

5.22. Выберите структуру данных и напишите программу вычисления множеств PLACES (определены в разделе 5.6), которая выполнялась бы за среднее время порядка $O(n)$ для строк длиной *n*.

***5.23.** Измените процедуру НОП из листинга **5.15** так, чтобы она действительно вычисляла НОП.

***5.24.** Напишите программу SPLIT для работы с 2-3 деревьями.

***5.25.** Пусть элементы множества, представленного посредством 2-3 дерева, содержат только поле ключей, а элементы, которые хранятся во внутренних узлах, не имеют представительства в листьях. С учетом этих предположений перепишите операторы словарей, избегая хранения каких-либо элементов в двух различных узлах.

Библиографические примечания

Нагруженные деревья (trie) впервые предложены в работе [38]. В [6] введены В-деревья (с которыми мы познакомимся в главе 11) как обобщения 2-3 деревьев. Впервые использовали деревья 2-3 для вставки и удаления элементов и для слияния и разбиения множеств Дж. Хопкрофт (J. E. Hopcroft) в 1970 г. (не опубликовано), а для оптимизации кодов — Дж. Ульман (J. D. Ullman) [111].

Структуры деревьев из раздела 5.5, использующие сжатие путей и слияние меньших деревьев в большие, впервые применили М. Мак-Илрой (М. D. McIlroy) и Р. Моррис (R. Morris) для построения остовных деревьев минимальной стоимости. Реа-

лизация АТД MFSET посредством деревьев проанализирована в работах [31] и [53]. Упражнение 5.21,б взято из статьи [108].

Решение задачи НОП из раздела 5.6 приведено в [55]. Эффективная структура данных для операторов FIND, SPLIT и суженного оператора MERGE (когда все элементы в одном множестве меньше по величине, чем в другом) приведена в статье [113].

Упражнение 5.6 основано на эффективном алгоритме сравнения шаблонов, разработанном в [116]. Вариант 2-3 деревьев из упражнения 5.16 подробно рассмотрен в монографии [126]. Структура АВЛ-деревьев для упражнения 5.18 взята из работы [1].

ГЛАВА 6

Ориентированные графы

Во многих задачах, встречающихся в компьютерных науках, математике, технических **дисциплинах** часто возникает необходимость наглядного представления отношений между какими-либо объектами. Ориентированные и **неориентированные** графы — естественная модель для таких отношений. В этой главе рассмотрены основные структуры данных, которые применяются для представление ориентированных графов, а также описаны некоторые основные алгоритмы определения связности ориентированных графов и нахождения кратчайших **путей.**

6.1. Основные определения

Ориентированный граф (или сокращенно *орграф)* $G = (V, E)$ состоит из множества вершин V и множества дуг E. Вершины также называют *узлами,* а дуги — *ориентированными ребрами.* Дуга представима в виде упорядоченной пары вершин (v, w), где вершина v называется *началом,* а w — *концом* дуги. Дугу (v, w) часто записывают как $v \to w$ и изображают в виде



Говорят также, что дуга $v \to w$ *ведет от вершины v к вершине w,* а вершина w *смежная* с вершиной v.

Пример 6.1. На рис. 6.1 показан орграф с четырьмя вершинами и пятью дугами. D

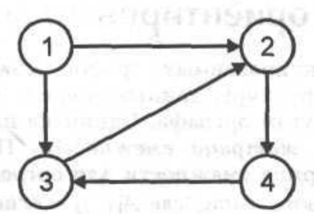


Рис. 6.1. Ориентированный граф

Вершины орграфа можно использовать для представления объектов, а дуги — для отношений между объектами. Например, вершины орграфа могут представлять города, а дуги — маршруты рейсовых полетов самолетов из одного города в другой. В другом примере, рассмотренном нами ранее в разделе 4.2, в виде орграфа представлена блок-схема потока данных в компьютерной программе. В последнем примере вершины соответствуют блокам операторов программы, а дугам — направленное перемещение потоков данных.

Путем в орграфе называется последовательность вершин $v_1, v_2, ..., v_n$, для которой существуют дуги $v_1 \to v_2$, $v_2 \to v_3$, ..., $v_{n-1} \to v_n$. Этот путь начинается в вершине v_1 и, проходя через вершины $v_2, v_3, \ldots, v_{n-1}$, заканчивается в вершине v_n. *Длина пу-*

ти — количество дуг, составляющих путь, в данном случае длина пути равна л - 1. Как осрбый случай пути рассмотрим одну вершину *v* как путь длины О от вершины *v* к этой же вершине *v*. На рис. 6.1 последовательность вершин 1, 2, 4 образуют путь длины 2 от вершины 1 к вершине 4.

Путь называется *простым,* если все вершины на нем, за исключением, может быть, первой и последней, различны. *Цикл* — это простой путь длины не менее **1,** который начинается и заканчивается в одной и той же вершине. На рис. 6.1 вершины 3, 2, 4, 3 образуют цикл длины 3.

Во многих приложениях удобно к вершинам и дугам орграфа присоединить какую-либо информацию. Для этих целей используется *помеченный орграф,* т.е. орграф, у которого каждая дуга и/или каждая вершина имеет соответствующие метки. Меткой может быть имя, вес или стоимость (дуги), или значение данных какого-либо заданного типа.

Пример 6.2. На рис. 6.2 показан помеченный орграф, в котором каждая дуга имеет буквенную метку. Этот орграф имеет интересное свойство: любой цикл, начинающийся в вершине 1, порождает последовательность букв *а* и *Ь,* в которой всегда четное количество букв а и ***b***. О

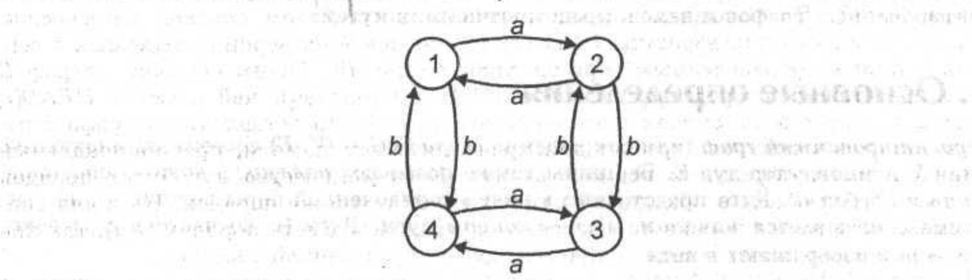


Рис. 6.2. Помеченный орграф

В помеченном орграфе вершина может иметь как имя, так и метку. Часто метки вершин мы будем использовать в качестве имен вершин. Так, на рис. 6.2 числа, помещенные в вершины, можно интерпретировать как имена вершин и как метки вершин.

6.2. Представления ориентированных графов

Для представления ориентированных графов можно использовать различные структуры данных. Выбор структуры данных зависит от операторов, которые будут применяться к вершинам и дугам орграфа. Одним из наиболее общих представлений орграфа G = *(V, E)* является *матрица смежности.* Предположим, что множество вершин *V* = {1, 2, ..., *n).* Матрица смежности для орграфа *G* — это матрица *А* размера *n* х *n* со значениями булевого типа, где *A[i, j]* = true тогда и только тогда, когда существует дуга из вершины *i в* вершину *j.* Часто в матрицах смежности значение true заменяется на 1, а значение false — на 0. Время доступа к элементам матрицы смежности зависит от размеров множества вершин и множества дуг. Представление орграфа в виде матрицы смежности удобно применять в тех алгоритмах, в которых надо часто проверять существование данной дуги.

Обобщением описанного представления орграфа с помощью матрицы смежности можно считать представление помеченного орграфа также посредством матрицы смежности, но у которой элемент *A[i, j]* равен метке дуги *i* → *j*. **Если** дуги от вершины i к вершине *j* не существует, то значение *A[i, j]* не может быть значением какой-либо допустимой метки, а может рассматриваться как "пустая" ячейка.

Пример 6.3. На рис. 6.3 показана матрица смежности для помеченного орграфа из рис. 6.2. Здесь дуги представлены меткой символьного типа, а пробел используется при отсутствии дуг. П

Основной недостаток матриц смежности заключается в том, что она требует $\Omega(n^2)$ объема памяти, даже если дуг значительно меньше, чем n^2. Поэтому для чтения матрицы или нахождения в ней необходимого элемента требуется время порядка $O(n^2)$, что не позволяет создавать алгоритмы с временем $O(n)$ для работы с орграфами, имеющими порядка $O(n)$ дуг.

	1	2	3	4
1		*a*		*b*
2	*a*		*b*	
3		*b*		*a*
4	*b*		*a*	

Рис. 6.3. Матрица смежности для помеченного орграфа из рис. 6.2

Поэтому вместо матриц смежности часто используется другое представление для орграфа G = (V, *E),* называемое представлением посредством *списков смежности.* Списком смежности для вершины *i* называется список всех вершин, смежных с вершиной i, причем определенным образом упорядоченный. Таким образом, орграф G можно представить посредством массива *HEAD* (Заголовок), чей элемент *HEAD*[*i*] является указателем на список смежности вершины i. Представление орграфа с помощью списков смежности требует для хранения объем памяти, пропорциональный сумме количества вершин и количества дуг. Если количество дуг имеет порядок $O(n)$, *то* и общий объем необходимой памяти имеет такой же порядок. Но и для списков смежности время поиска определенной дуги может иметь порядок $O(n)$, так как такой же порядок может иметь количество дуг у определенной вершины.

Пример 6.4. На рис. 6.4 показана структура данных, представляющая орграф из рис. 6.1 посредством связанных списков смежности. Если дуги имеют метки, то их можно хранить в ячейках связанных списков.

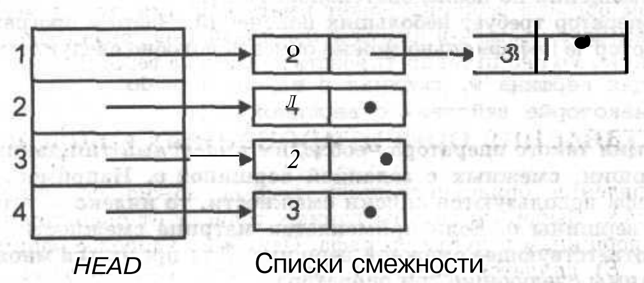


Рис. 6.4. Структура списков смежности для орграфа из рис. 6.1

Для вставки и удаления элементов в списках смежности необходимо иметь массив *HEAD,* содержащий указатель на ячейки заголовков списков смежности, но не сами смежные вершины[1]. Но если известно, что граф не будет подвергаться изменениям (или они будут незначительны), то предпочтительно сделать массив *HEAD* массивом курсоров на массив *ADJ* (от adjacency — смежность), где ячейки *ADJ*[*HEAD*[*i*]], *ADJ*[*HEAD*[*i*] + 1] и т.д. содержат вершины, смежные с вершиной *i,* и эта последовательность смежных вершин заканчивается первым встреченным нулем в массиве *ADJ.* Пример такого представления для графа из рис. 6.1 показан на рис. 6.5. D

[1] Здесь проявляется старая проблема языка Pascal: сложность выполнения вставки и удаления произвольных позиций в простых связанных списках.

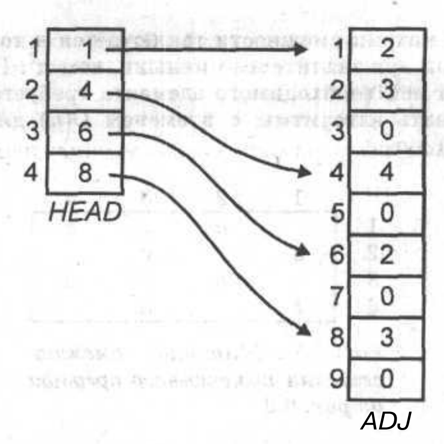


Рис. 6.5. Другое представление списков смежности для графа из рис. 6.1

АТД для ориентированных графов

По привычной схеме сначала надо формально определить абстрактные типы данных, соответствующие **ориентированным** графам, а затем рассмотреть операторы, выполняемые над этими АТД. Но мы не будем неукоснительно следовать этой схеме, так как на этом пути нам не встретятся большие неожиданности, а принципиальные структуры данных, используемые для представления орграфов, уже были описаны ранее. Наиболее общие операторы, выполняемые над ориентированными графами, **включают** операторы чтения меток вершин и дуг, вставки и удаления вершин и дуг и оператор перемещения по последовательностям дуг.

Последний оператор требует небольших пояснений. Часто в программах встречаются операторы, которые неформально можно описать подобно следующему оператору:

```
for каждая вершина w, смежная с вершиной v do
    { некоторые действия с вершиной w }                    (6.1)
```

Для реализации такого оператора необходим *индексный* тип данных для работы с множеством вершин, смежных с заданной вершиной v. Например, если для представления орграфа используются списки смежности, то индекс — это позиция в списке смежности вершины v. Если применяется матрица смежности, тогда индекс — целое число, соответствующее смежной вершине. Для просмотра множества смежных вершин необходимы следующие три оператора.

1. **FIRST**(v) возвращает индекс первой вершины, смежной с вершиной v. Если вершина v не имеет смежных вершин, то возвращается "нулевая" вершина Л.
2. **NEXT**(v, i) возвращает индекс вершины, смежной с вершиной v, следующий за индексом i. Если i — это индекс последней вершины, смежной с вершиной v, то возвращается Л.
3. **VERTEX**(v, i) возвращает вершину с индексом i из множества вершин, смежных с v.

Пример 6.5. Если для представления орграфа используется матрица смежности, то функция **VERTEX**(v, i) просто возвращает индекс i. Реализация функций **FIRST**(v) и **NEXT**(v, i) представлена в листинге 6.1. В этом листинге матрица смежности A размера n x n определена вне этих функций с помощью следующего объявления:

```
array [1..n, 1..n] of boolean
```

В этой матрице 0 обозначает отсутствие дуги. Эти функции позволяют реализовать оператор (6.1) так, как показано в листинге 6.2. Отметим, что функцию FIRST(v) можно реализовать как NEXT(v, 0). □

Листинг 6.1. Операторы просмотра множества смежных вершин

```
function FIRST ( v: integer ): integer;
    var
        i: integer;
    begin
        for i:= 1 to n do
            if A[v, i] then
                return(i);
        return(0) { вершина v не имеет смежных вершин}
    end; { FIRST }

function NEXT ( v: integer; i: integer): integer;
    var
        j: integer;
    begin
        for j:= i + 1 to n do
            if A[v, j] then
                return(j);
        return(0)
    end; { NEXT }
```

Листинг 6.2. Последовательный просмотр вершин, смежных с вершиной v

```
i:= FIRST(v);
while i <> 0 do begin
    w:= VERTEX(v, i);
    { действия с вершиной w }
    i:= NEXT(v, i)
end
```

6.3. Задача нахождения кратчайшего пути

В этом разделе мы рассмотрим задачи нахождения путей в ориентированном графе. Пусть есть ориентированный граф $G = (V, E)$, у которого все дуги имеют неотрицательные метки (стоимости дуг), а одна вершина определена как *источник*. Задача состоит в нахождении стоимости кратчайших путей от источника ко всем другим вершинам графа G (здесь *длина пути* определяется как сумма стоимостей дуг, составляющих путь). Эта задача часто называется *задачей нахождения кратчайшего пути с одним источником*[1]. Отметим, что мы будем говорить о длине пути даже тогда, когда она измеряется в других, не линейных, единицах измерения, например во временных единицах.

Можно представить орграф G в виде карты маршрутов рейсовых полетов из одного города в другой, где каждая вершина соответствует городу, а дуга $v \to w$ — рейсовому маршруту из города v в город w. Метка дуги $v \to w$ — это время полета из горо-

[1] Может показаться, что более естественной задачей будет нахождение кратчайшего пути от источника к определенной вершине *назначения*. Но эта задача в общем случае имеет такой же уровень сложности, что и задача нахождения кратчайшего пути для всех вершин графа (за исключением того "счастливого" случая, когда путь к вершине назначения будет найден ранее, чем просмотрены пути ко всем вершинам графа).

да v в город w[1]. В этом случае решение задачи нахождения кратчайшего пути с одним источником для ориентированного графа трактуется как минимальное время перелетов между различными городами.

Для решения поставленной задачи будем использовать "жадный" (см. раздел 1.1) алгоритм, который часто называют *алгоритмом Дейкстры* (Dijkstra). Алгоритм строит множество S вершин, для которых кратчайшие пути от источника уже известны. На каждом шаге к множеству S добавляется та из оставшихся вершин, расстояние до которой от источника меньше, чем для других оставшихся вершин. Если стоимости всех дуг неотрицательны, то можно быть уверенным, что кратчайший путь от источника к конкретной вершине проходит только через вершины множества S. Назовем такой путь *особым*. На каждом шаге алгоритма используется также массив D, в который записываются длины кратчайших особых путей для каждой вершины. Когда множество S будет содержать все вершины орграфа, т.е. для всех вершин будут найдены "особые" пути, тогда массив D будет содержать длины кратчайших путей от источника к каждой вершине.

Алгоритм Дейкстры представлен в листинге 6.3. Здесь предполагается, что в орграфе G вершины поименованы целыми числами, т.е. множество вершин $V = \{1, 2, ..., n\}$, причем вершина 1 является источником. Массив C — это двумерный массив стоимостей, где элемент $C[i, j]$ равен стоимости дуги $i \to j$. Если дуги $i \to j$ не существует, то $C[i, j]$ ложится равным ∞, т.е. большим любой фактической стоимости дуг. На каждом шаге $D[i]$ содержит длину текущего кратчайшего особого пути к вершине i.

Листинг 6.3. Алгоритм Дейкстры

```
        procedure Dijkstra;
           begin
(1)           S:= {1};
(2)           for i:= 2 to n do
(3)              D[i]:= C[1, i]; { инициализация D }
(4)           for i:= 1 to n - 1 do begin
(5)              выбор из множества V\S такой вершины w,
                    что значение D[w] минимально;
(6)              добавить w к множеству S;
(7)              for каждая вершина v из множества V\S do
(8)                 D[v] := min(D[v] , D[w] + C[w, v])
              end
           end; { Dijkstra }
```

Пример 6.6. Применим алгоритм Дейкстры для ориентированного графа, показанного на рис. 6.6. Вначале $S = \{1\}$, $D[2] = 10$, $D[3] = \infty$, $D[4] = 30$ и $D[5] = 100$. На первом шаге цикла (строки (4) - (8) листинга 6.3) $w = 2$, т.е. вершина 2 имеет минимальное значение в массиве D. Затем вычисляем $D[3] = \min(\infty, 10 + 50) = 60$. $D[4]$ и D[5] не изменяются, так как не существует дуг, исходящих из вершины 2 и ведущих к вершинам 4 и 5. Последовательность значений элементов массива D после каждой итерации цикла показаны в табл. 6.1. D

Несложно внести изменения в алгоритм так, чтобы можно было определить сам кратчайший путь (т.е. последовательность вершин) для любой вершины. Для этого надо ввести еще один массив P вершин, где $P[v]$ содержит вершину, непосредственно предшествующую вершине v в кратчайшем пути. Вначале положим $P[v] = 1$ для всех $v \neq 1$. В листинге 6.3 после строки (8) надо записать условный оператор с условием

[1] Можно предположить, что в данном случае в качестве модели больше подходит неориентированный граф, поскольку метки дуг $v \to w$ и $w \to v$ могут совпадать. Но фактически в большинстве случаев время полета в противоположных направлениях между двумя городами различно. Кроме того, предположение о совпадении меток дуг $v \to w$ и $w \to v$ не влияет (и не помогает) на решение задачи нахождения кратчайшего пути.

$D[w] + C[w, v] < D[v]$, при выполнении которого элементу $P[v]$ присваивается значение w. После выполнения алгоритма кратчайший путь к каждой вершине можно найти с помощью обратного прохождения по предшествующим вершинам массива P.

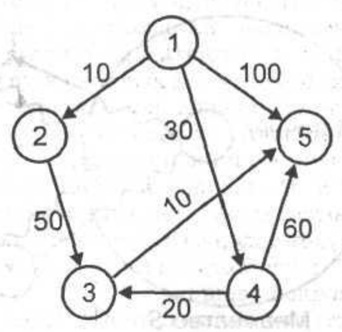


Рис. 6.6. Орграф с помеченными дугами

Таблица 6.1. Вычисления **по алгоритму Дейкстры для орграфа из рис.** 6.6

Итерация	S	w	$D[2]$	$D[3]$	$D[4]$	fi[5]
Начало	{1}	–	10	∞	30	100
1	{1, 2}	2	10	60	30	100
2	<1, 2, 4}	4	10	50	30	90
3	{1, 2, 4, 3}	3	10	50	30	60
4	{1, 2, 4, 3, 5}	5	10	50	30	60

Пример 6.7. Для орграфа из примера 6.6 массив P имеет следующие значения: $P[2] = 1$, $P[3] = 4$, $P[4] = 1$, $P[5] = 3$. Для определения кратчайшего пути, например от вершины 1 к вершине 5, надо отследить в обратном порядке предшествующие вершины, начиная с вершины 5. На основании значений массива P определяем, что вершине 5 предшествует вершина 3, вершине 3 — вершина 4, а ей, в свою очередь, — вершина 1. Таким образом, кратчайший путь из вершины 1 в вершину 5 составляет следующая последовательность вершин: 1, 4, 3, 5. □

Обоснование алгоритма Дейкстры

Алгоритм Дейкстры — пример алгоритма, где "жадность" окупается в том смысле, что если что-то "хорошо" локально, то оно будет "хорошо" и глобально. В данном случае что-то локально "хорошее" — это вычисленное расстояние от источника к вершине w, которая пока не входит в множество S, но имеет кратчайший особый путь. (Напомним, что особым мы назвали путь, который проходит только через вершины множества S.) Чтобы понять, почему не может быть другого кратчайшего, но не особого, пути, рассмотрим рис. 6.7. Здесь показан гипотетический кратчайший путь к вершине w, который сначала проходит до вершины x через вершины множества S, затем после вершины x путь, возможно, несколько раз входит в множество S и выходит из него, пока не достигнет вершины w.

Но если этот путь короче кратчайшего особого пути к вершине w, то и начальный сегмент пути от источника к вершине x (который тоже является особым путем) также короче, чем особый путь к w. Но в таком случае в строке (5) листинга 6.3 при выборе вершины w мы должны были выбрать не эту вершину, а вершину x, поскольку $D[x]$ меньше $D[w]$. Таким образом, мы пришли к противоречию, следовательно, не может

быть другого кратчайшего пути к вершине *w*, кроме особого. (Отметим здесь определяющую роль того факта, что все стоимости дуг неотрицательны, без этого свойства помеченного орграфа алгоритм Дейкстры не будет работать правильно.)

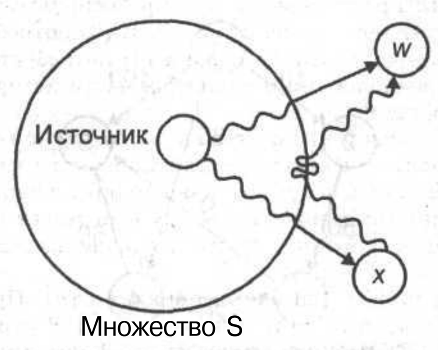


Рис. 6.7. ***Гипотетический** кратчайший **путь** к вершине w*

Для завершения обоснования алгоритма Дейкстры надо еще доказать, что *D[v]* действительно показывает кратчайшее расстояние до вершины *v*. Рассмотрим добавление новой вершины *w* к множеству *S* (строка (6) листинга 6.3), после чего происходит пересчет элементов массива *D* в строках (7), (8) этого листинга; при этом может получиться более короткий особый путь к некоторой вершине *v*, проходящий через вершину *w*. Если часть этого пути до вершины *w* проходит через вершины предыдущего множества *S* и затем непосредственно к вершине *v*, то стоимость этого пути, *D*[*w*] + *C*[*w*, *v*], *в* строке (8) листинга 6.3 сравнивается с *D*[*v*] и, если новый путь короче, изменяется значение *D[v]*.

Существует еще одна гипотетическая возможность кратчайшего особого пути к вершине *v*, которая показана на рис. 6.8. Здесь этот путь сначала идет к вершине *w*, затем возвращается к вершине *x*, принадлежащей предыдущему множеству *S*, затем следует к вершине *v*. Но реально такого кратчайшего пути не может быть. Поскольку вершина *x* помещена в множество *S* раньше вершины *w*, то все кратчайшие пути от источника к вершине *x* проходят исключительно через вершины предыдущего множества *S*. Поэтому показанный на рис. 6.8 путь к вершине *x*, проходящий через вершину *w*, не короче, чем путь к вершине *x*, проходящий через вершины множества *S*. В результате и весь путь к вершине *v*, проходящий через вершины *x* и *w*, не короче, чем путь от источника к вершине *x*, проходящий через вершины множества *S*, и далее непосредственно к вершине *v*. Таким образом, доказано, что оператор в строке (8) листинга 6.3 действительно вычисляет длину кратчайшего пути.

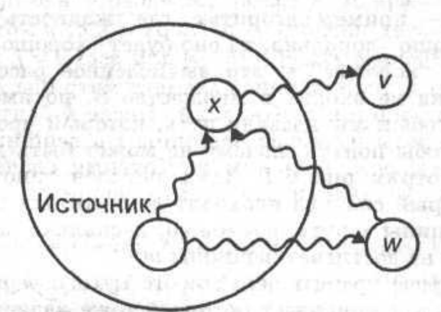


Рис. 6.8. Реально невозможный кратчайший особый путь

Время выполнения алгоритма Дейкстры

Предположим, что процедура листинга **6.3** оперирует с орграфом, имеющим n вершин и e дуг. Если для представления орграфа используется матрица смежности, то для выполнения внутреннего цикла строк (7) и (8) потребуется время $O(n)$, а для выполнения всех n - 1 итераций цикла строки (4) потребуется время порядка $O(n^2)$. Время, необходимое для выполнения оставшейся части алгоритма, как легко видеть, не превышает этот же порядок.

Если количество дуг e значительно меньше n^2, то лучшим выбором для представления орграфа будут списки смежности, а для множества вершин $V \setminus S$ — очередь с приоритетами, реализованная в виде частично упорядоченного дерева. Тогда время выбора очередной вершины из множества $V \setminus S$ и пересчет стоимости путей для одной дуги составит $O(\log n)$, а общее время выполнения цикла строк (7) и (8) — $O(e \log n)$, а не $O(n^2)$.

Строки (1) - (3) выполняются за время порядка $O(n)$. При использовании очередей с приоритетом для представления множества $V \setminus S$ строка (5) реализуется посредством оператора DELETEMIN, а каждая из n - 1 итераций цикла (4) - (6) требует времени порядка $O(\log n)$.

В результате получаем, что общее время выполнения алгоритма Дейкстры ограничено величиной порядка $O(c \log n)$. Это время выполнения значительно меньше, чем $O(n^2)$, когда e существенно меньше n^2.

6.4. Нахождение кратчайших путей между парами вершин

Предположим, что мы имеем помеченный орграф, который содержит время полета по маршрутам, связывающим определенные города, и мы хотим построить таблицу, где приводилось бы минимальное время перелета из одного (произвольного) города в любой другой. В этом случае мы сталкиваемся с *общей задачей нахождения кратчайших путей,* т.е. нахождения кратчайших путей между всеми парами вершин орграфа. Более строгая формулировка этой задачи следующая: есть ориентированный граф G = (V, E), каждой дуге $v \to w$ этого графа сопоставлена неотрицательная стоимость $C[v, w]$. Общая задача нахождения кратчайших путей заключается в нахождении для каждой упорядоченной пары вершин (v, w) любого пути от вершины v в вершины w, длина которого минимальна среди всех возможных путей от v к w.

Можно решить эту задачу, последовательно применяя алгоритм Дейкстры для каждой вершины, объявляемой в качестве источника. Но существует прямой способ решения данной задачи, использующий *алгоритм Флойда* (R. W. Floyd). Для опрёт деленности положим, что вершины графа последовательно пронумерованы от **1** до n. Алгоритм Флойда использует матрицу A размера л х л, в которой вычисляются длины кратчайших путей. Вначале $A[i, j] = C[i, j]$ для всех $i \neq j$. Если дуга $i \to j$ отсутствует, то $C[i, j] = \infty$. Каждый диагональный элемент матрицы A равен 0.

Над матрицей A выполняется л итераций. После k-й итерации $A[i, j]$ содержит значение наименьшей длины путей из вершины i в вершину j, которые не проходят через вершины с номером, большим k. Другими словами, между концевыми вершинами пути i и j могут находиться только вершины, номера которых меньше или равны k.

На fe-й итерации для вычисления матрицы А применяется следующая формула:

$$A_k[i, j] = \min(A_{k-1}[i, j], A_{k-1}[i, k] + A_{k-1}[k, j]).$$

Нижний индекс k обозначает значение матрицы A после fe-й итерации, но это не означает, что существует л различных матриц, этот индекс используется для сокращения записи. Графическая интерпретация этой формулы показана на рис. 6.9.

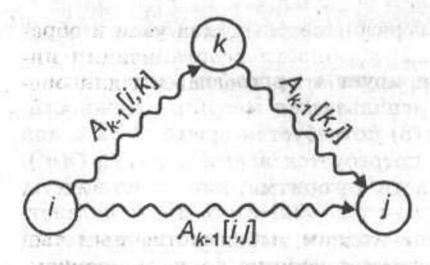


Рис. 63. Включение вершины k в путь от вершины i к вершине j

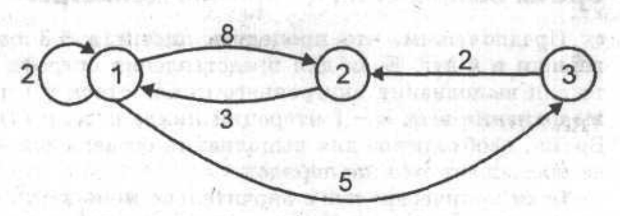
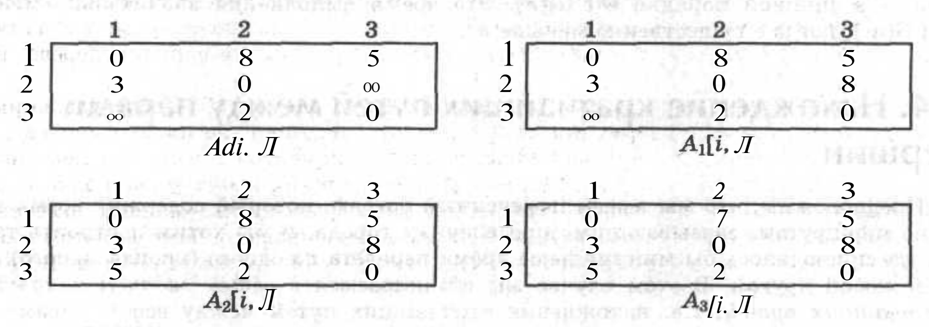


Рис. 6.10. Помеченный ориентированный граф

Для вычисления $A_k[i, j]$ проводится сравнение величины $A_{k-1}[i, j]$ (т.е. стоимость пути от вершины *i* к вершине *j* без участия вершины *k или* другой вершины с более высоким номером) с величиной $A_{k-1}[i, k] + A_{k-1}[k, f]$ (стоимость пути от вершины i до вершины А плюс стоимость пути от вершины А до вершины *j*). Если путь через вершину *k* дешевле, чем $A_{k-1}[i, f]$, то величина $A_k[i, f]$ изменяется.

Пример 6.8. На рис. 6.10 показан помеченный орграф, а на рис. 6.11 — значения матрицы *А* после трех итераций. □



Adi. Л

	1	2	3
1	0	8	5
2	3	0	∞
3	∞	2	0

A_1[i, Л

	1	2	3
1	0	8	5
2	3	0	8
3	∞	2	0

A_2[i, Л

	1	2	3
1	0	8	5
2	3	0	8
3	5	2	0

A_3[i. Л

	1	2	3
1	0	7	5
2	3	0	8
3	5	2	0

Рис. 6.11. Последовательные значения матрицы А

Равенства $A_k[i, А] = A_{k-1}[i, А]$ и $A_k[k, j] = A_{k-1}[k, у]$ означают, что на А-й итерации элементы матрицы *А*, стоящие в А-й строке и А-м столбце, не изменяются. Более того, все вычисления можно выполнить с применением только одной копии матрицы *А*. Процедура, реализующая алгоритм Флойда, представлена в следующем листинге.

Листинг 6.4. Реализация алгоритма Флойда

```
procedure Floyd { var A: array[1..n, 1..n] of real;
                      C: array[1..n, 1..n] of real);
    var
        i, j, k: integer;
    begin
        for i:= 1 to л do
            for j:= 1 to n do
                A[i, j]:= C[i, j];
        for i:= 1 to л do
            A[i, i]:= 0;
        for k:= 1 to л do
            for i:= 1 to л do
                for j:= 1 to л do
                    if A[i, k] + A[k, j] < A(i, j] then
                        A[i, j]:= A[i, k] + A[k, j]
    end; { Floyd }
```

Время выполнения этой программы, очевидно, имеет порядок $O(n^3)$, поскольку в ней практически нет ничего, кроме вложенных друг в друга трех циклов. Доказательство "правильности" работы этого алгоритма также очевидно и выполняется с помощью математической индукции по *k*, показывая, что на ft-й итерации вершина *k* включается в путь только тогда, когда новый путь короче старого.

Сравнение алгоритмов Флойда и Дейкстры

Поскольку версия алгоритма Дейкстры с использованием матрицы смежности находит кратчайшие пути от одной вершины за время порядка $O(n^2)$, то в этом случае применение алгоритма Дейкстры для нахождения всех кратчайших путей потребует времени порядка $O(n^3)$, т.е. получим такой же временной порядок, как и в алгоритме Флойда. Константы пропорциональности в порядках времени выполнения для обоих алгоритмов зависят от применяемых компилятора и вычислительной машины, а также от особенностей реализации алгоритмов. Вычислительный эксперимент и измерение времени выполнения — самый простой путь подобрать лучший алгоритм для конкретного приложения.

Если *e*, количество дуг в орграфе, значительно меньше, чем n^2, тогда, несмотря на относительно малую константу в выражении порядка $O(n^3)$ для алгоритма Флойда, мы рекомендуем применять версию алгоритма Дейкстры со списками смежности. В этом случае время решения общей задачи нахождения кратчайших путей имеет порядок $O(ne \log n)$, что значительно лучше алгоритма Флойда, по крайней мере для больших разреженных графов.

Вывод на печать кратчайших путей

Во многих ситуациях требуется распечатать самый дешевый путь от одной вершины к другой. Чтобы восстановить при необходимости кратчайшие пути, можно в алгоритме Флойда ввести еще одну матрицу *P*, в которой элемент *P[i, j]* содержит вершину *k*, полученную при нахождении наименьшего значения *A[i, j]*. Если *P[i, j]* = O, то кратчайший путь из вершины *i* в вершину *j* состоит из одной дуги $i \to j$. Модифицированная версия алгоритма Флойда, позволяющая восстанавливать кратчайшие пути, представлена в листинге 6.5.

Листинг 6.5. Программа нахождения кратчайших путей

```
procedure Floyd ( var A: array[1..n, 1..n] of real;
        C:array[1..n, 1..n] of real; P:array[1..n, 1..n] of integer);
        var
            i, j, k: integer;
        begin
            for i:= 1 to n do
                for j:= 1 to n do begin
                    A[i, j]:=C[i, j];
                    P[i, j]:= 0
                end;
            for i:= 1 to л do
                A[i, i]:= 0;
            for k:= \ to n do
                for i:= 1 to n do
                    for j:= 1 to л do
                        if A[i, k] + A[k, j] < A[i, j] then begin
                            A[i, j]:= A[i, k] + A[k, j];
                            P(i, j]:= k
                        end
        end; { Floyd }
```

Для вывода на печать последовательности вершин, составляющих кратчайший путь от вершины i до вершины j, вызывается процедура **path**(i, j), код которой приведен в листинге 6.6.

Листинг 6.6. Процедура печати кратчайшего пути

```
procedure path ( i, j: integer );
    var
        k: integer;
    begin
        k:= P[i, j];
        if k = 0 then
            return;
        path(i, k);
        writeln(k);
        path(k, j)
    end; { path }
```

Пример 6.9. На рис. 6.12 **показана** результирующая матрица P для орграфа из рис. 6.10. D

	1	2	3
1	0	3	0
2	0	0	1
3	2	0	0

P

Рис. 6.12. Матрица P для орграфа из рис. 6.10

Транзитивное замыкание

Во многих задачах интерес представляет только сам факт существования пути, длиной не меньше единицы, от вершины i до вершины j. Алгоритм Флойда можно приспособить для решения таких задач. Но полученный в результате алгоритм еще до Флойда разработал Уоршелл (S. Warshall), поэтому мы будем называть его алгоритмом Уоршелла.

Предположим, что матрица стоимостей C совпадает с матрицей смежности для данного орграфа G, т.е. $C[i, j] = 1$ только в том случае, если есть дуга $i \to j$, и $C[i,$ Л ~ 0, если такой дуги не существует. Мы хотим вычислить матрицу A такую, что $A[i, j] = 1$ тогда и только тогда, когда существует путь от вершины i до вершины j длиной не менее 1 и $A[i, f] = 0$ — в противном случае. Такую матрицу A часто называют *транзитивным замыканием* матрицы смежности.

Пример 6.10. На рис. 6.13 показано транзитивное замыкание матрицы смежности орграфа из рис. **6.10.** D

	1	2	3
1	1	1	1
2	1	1	1
3	1	1	1

Рис. 6.13. Транзитивное замыкание матрицы смежности

Транзитивное замыкание можно вычислить с помощью процедуры, подобной *Floyd,* применяя на fe-м шаге следующую формулу к булевой матрице A:

$$A_k[i, j] = A_{k-1}[i, f] \text{ or } (A_{k-1}[i, \text{А}] \text{ and } A_{k-1}[k, j]).$$

Эта формула устанавливает, что существует путь от вершины i до вершины j, проходящий через вершины с номерами, не превышающими k, только в следующих случаях.

1. Уже существует путь от вершины i до вершины j, который проходит через вершины с номерами, не превышающими k - 1.
2. Существует путь от вершины i до вершины k, проходящий через вершины с номерами, не превышающими k - 1, и путь от вершины k до вершины j, который также проходит через вершины с номерами, не превышающими k — 1.

Здесь, как и в алгоритме **Флойда,** $A_k[i, k] = A_{k-1}[i, k]$ и $A_k[k, j] = A_{k-1}[k,$ Л. **и вы**числения можно выполнять в одной копии матрицы A. Программа ***Warshall*** вычисления транзитивного замыкания **показана** в листинге 6.7.

Листинг 6.7. Программа ***Warshall* для вычисления транзитивного замыкания**

```
procedure Warshall ( var A: array[1..n, 1..n] of boolean;
          C: array[1..n, 1..n] of boolean );
     var
          i, j,  k: integer;
     begin
          for i:= 1 to л do
               for j:= 1 to л do
                    A[i, j]:= C[i, j];
          for k:= 1 to л do
               for i:= 1 to л do
                    for j:= 1 to л do
                         if A/i, j] = false then
                              A/i, j]:= A[i, k] and A(k, j]
     end; { Warshall }
```

Нахождение центра ориентированного графа

Предположим, что необходимо найти центральную вершину в орграфе. Эту задачу также можно решить с помощью алгоритма Флойда. Но сначала надо уточнить термин *центральная вершина.* Пусть v — произвольная вершина орграфа G = (V, E). ***Эксцентриситет***[1] вершины v определяется как

$$\max_{w \in V}\{\text{минимальная длина пути от вершины } w \text{ до вершины } v\}$$

Центром ***орграфа*** *G* называется вершина с минимальным эксцентриситетом. Другими **словами,** центром орграфа является вершина, для **которой** максимальное расстояние (длина пути) до других вершин минимально.

Пример **6.11.** Рассмотрим помеченный орграф, показанный на рис. 6.14.

В этом графе вершины имеют следующие эксцентриситеты.

Вершива	**Эксцентриситет**
a	∞
Ь	6
c	8
d	5
e	7

Из этой таблицы видно, что центром данного орграфа является вершина d. □

[1] **В русской математической литературе наряду с термином "эксцентриситет" часто используется термин "максимальное удаление". —** ***Прим. ред.***

Найти центр орграфа сравнительно просто. Пусть C — матрица стоимостей для орграфа G.

1. Сначала применим процедуру *Floyd* (листинг 6.4) к матрице C для вычисления матрицы *A,* содержащей все кратчайшие пути орграфа *G.*
2. Находим максимальное значение в каждом столбце *i* матрицы *A.* Это значение равно эксцентриситету вершины *i.*
3. Находим вершину с минимальным эксцентриситетом. Она и будет центром графа *G.*

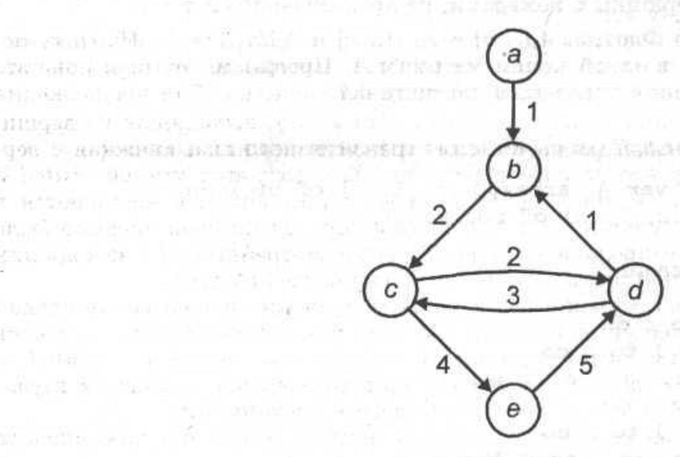


Рис. 6.14. Помеченный орграф

Время выполнения этого процесса определяется первым шагом, для которого время имеет порядок $O(n^3)$. Второй шаг требует времени порядка $O(n^2)$, а третий — $O(n)$.

Пример 6.12. Матрица всех кратчайших путей для орграфа из рис. 6.14 представлена на рис. 6.15. Максимальные значения в каждом столбце приведены под матрицей. D

	a	*b*	*c*	*d*	*e*
a	0	1	3	5	7
b	∞	0	2	4	6
c	∞	3	0	2	4
d	∞	1	3	0	7
e	∞	6	8	5	0
max	**∞**	**6**	**8**	**5**	**7**

Рис. 6.15. Матрица кратчайших путей

6.5. Обход ориентированных графов

При решении многих задач, касающихся ориентированных графов, необходим эффективный метод систематического обхода вершин и дуг орграфов. Таким методом является так называемый *поиск в глубину* — обобщение метода обхода дерева в прямом порядке. Метод поиска в глубину составляет основу многих других эффективных алгоритмов работы с графами. В последних двух разделах этой главы представлены различные алгоритмы, основанные на методе поиска в глубину.

Предположим, что есть ориентированный граф G, в котором первоначально все вершины помечены меткой *unvisited* (не посещалась). Поиск в глубину начинается с выбора начальной вершины *v* графа *G,* для этой вершины **метка** ***unvisited*** меняется на метку *visited* (посещалась). Затем для каждой вершины, смежной с вершиной *v* и которая не посещалась ранее, рекурсивно применяется поиск в глубину. Когда все вершины, которые можно достичь из вершины *v,* будут "удостоены" посещения, поиск заканчивается. **Если** некоторые вершины остались не посещенными, то выбирается одна из них и поиск повторяется. Этот процесс продолжается до тех пор, пока обходом не будут охвачены все вершины орграфа G.

Этот метод обхода вершин орграфа называется поиском в глубину, поскольку поиск не посещенных вершин идет в направлении вперед (вглубь) до тех пор, пока это возможно. Например, пусть x — последняя посещенная вершина. Для продолжения процесса выбирается какая-либо нерассмотренная дуга $x \to y$, выходящая из вершины x. Если вершина y уже посещалась, то ищется другая вершина, смежная с вершиной x. Если вершина y ранее не посещалась, то она помечается меткой *visited* и поиск начинается заново от вершины y. Пройдя все пути, которые начинаются в вершине y, возвращаемся в вершину x, т.е. в ту вершину, из которой впервые была достигнута вершина y. Затем продолжается выбор нерассмотренных дуг, исходящих из вершины x, и так до тех пор, пока не будут исчерпаны все эти дуги.

Для представления вершин, смежных с вершиной ***v*****,** можно использовать список смежности ***L*****[*****v*****]**, а для определения вершин, которые ранее посещались, — массив *mark* (метка), чьи элементы будут принимать только два значения: *visited* и *unvisited.* Эскиз рекурсивной процедуры *dfs* (от depth-first search — поиск в глубину), реализующей метод поиска в глубину, представлен в листинге 6.8.

Чтобы применить эту процедуру к графу, состоящему из *n* вершин, надо сначала присвоить всем элементам массива *mark* значение *unvisited,* затем начать поиск в глубину для каждой вершины, помеченной как *unvisited.* Описанное можно реализовать с помощью следующего кода:

```
for v:= 1 to n do
    mark[v]:= unvisited;
for v.= 1 to л do
    if mark[v] = unvisited then
        dfs(v)
```

Отметим, что листинг 6.8 является только эскизом процедуры, который еще следует детализировать. Заметим также, что эта процедура изменяет только значения массива *mark.*

Листинг 6.8. Процедура поиска в глубину

```
    procedure dfs ( v: вершина );
        var
            w: вершина;
        begin
(1)         mark[v]:= visited;
(2)         for каждая вершина w из списка L[v] do
(3)             if mark[w] = unvisited then
(4)                 dfs(w)
        end; { dfs }
```

Анализ процедуры поиска в глубину

Все вызовы процедуры *dfs* для полного обхода графа с *n* вершинами и *e* **дугами,** если $e > n$, требуют общего времени порядка $O(e)$. Чтобы показать это, заметим, что нет вершины, для которой процедура *dfs* вызывалась бы больше одного **раза,** поскольку рассматриваемая вершина помечается как *visited в* строке **(1)** (листинг 6.8)

еще до следующего вызова процедуры *dfs* и никогда не вызывается для вершин, помеченных этой меткой. Поэтому общее время выполнения строк (2) и (3) для просмотра всех списков смежности пропорционально сумме длин этих списков, т.е. имеет порядок *O(e)*. Таким образом, предполагая, что $e > n$, общее время обхода по всем вершинам орграфа имеет порядок *O(e)*, необходимое для "просмотра" всех дуг графа.

Пример 6.13. Пусть процедура *dfs* применяется к ориентированному графу, представленному на рис. 6.16, начиная с вершины *A*. Алгоритм помечает эту вершину как *visited* и выбирает вершину *B* из списка смежности вершины *A*. Поскольку вершина *B* помечена как *unvisited*, обход графа продолжается вызовом процедуры *dfs(B)*. Теперь процедура помечает вершину *B* как *visited* и выбирает первую вершину из списка смежности вершины *B*. В зависимости от порядка представления вершин в списке смежности, следующей рассматриваемой вершиной может быть или вершина *C*, или вершина *D*.

Предположим, что в списке смежности вершина C предшествует вершине *D*. Тогда осуществляется вызов *dfs(C)*. В списке смежности вершины C присутствует только вершина *A*, но она уже посещалась ранее. Поскольку все вершины в списке смежности вершины *C* исчерпаны, то поиск возвращается в вершину *B*, откуда процесс поиска продолжается вызовом процедуры *dfs(D)*. Вершины A и C из списка смежности вершины D уже посещались ранее, поэтому поиск возвращается сначала в вершину *B*, а затем в вершину *A*.

На этом первоначальный вызов *dfs(A)* завершен. Но орграф имеет вершины, которые еще не посещались: *E*, *F* и *G*. Для продолжения обхода вершин графа выполняется вызов *dfs(E)*. □

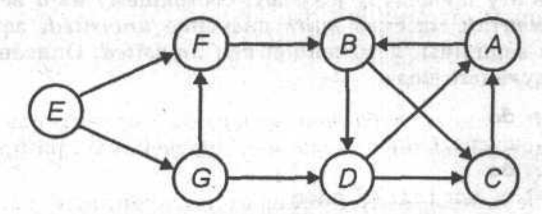


Рис. 6.16. Ориентированный граф

Глубинный остовный лес

В процессе обхода ориентированного графа методом поиска в глубину только определенные дуги ведут к вершинам, которые ранее не посещались. Такие дуги, ведущие к новым вершинам, называются *дугами дерева* и формируют для данного графа *остовный лес, построенный методом поиска в глубину,* или, сокращенно, *глубинный остовный лес*. На рис. 6.17 показан глубинный остовный лес для графа из рис. 6.16. Здесь сплошными линиями обозначены дуги дерева. Отметим, что дуги дерева формируют именно лес, т.е. совокупность деревьев, поскольку методом поиска в глубину к любой ранее не посещавшейся вершине можно придти только по одной дуге, а не по двум различным дугам.

В добавление к дугам дерева существуют еще три типа дуг, определяемых в процессе обхода орграфа методом поиска в глубину. Это обратные, прямые и поперечные дуги. *Обратные дуги* (как дуга $C \to A$ на рис. 6.17) — такие дуги, которые в остовном лесе идут от потомков к предкам. Отметим, что дуга из вершины в саму себя также является обратной дугой. *Прямыми дугами* называются дуги, идущие от предков к истинным потомкам, но которые не являются дугами дерева. На рис. 6.17 прямые дуги отсутствуют.

Дуги, такие как $D \to C$ и $G \to D$ на рис. 6.17, соединяющие вершины, не являющиеся ни предками, ни потомками друг друга, называются *поперечными дугами*. Ес-

ли при построении остовного дерева сыновья одной вершины располагаются слева направо в порядке их посещения и если новые деревья добавляются в лес также слева направо, то все поперечные дуги идут справа налево, что видно на рис. **6.17**. Такое взаимное расположение вершин и деревьев выбрано не случайно, так как это помогает формировать остовный лес.

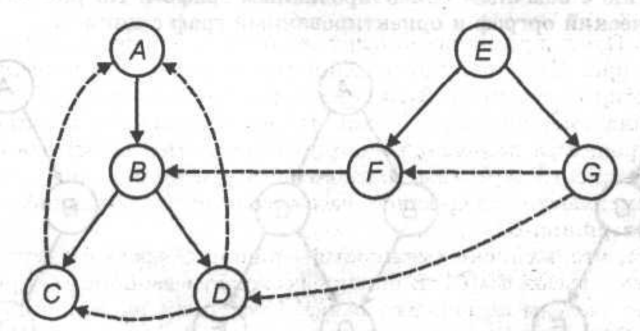


Рис. 6.17. Глубинный остовный лес для орграфа из рис. 6.16

Как можно различить эти четыре типа дуг? Легко определить дуги дерева, так как они получаются в процессе обхода графа как дуги, ведущие к тем вершинам, которые ранее не посещались. Предположим, что в процессе обхода орграфа его вершины нумеруются в порядке их посещения. Для этого в листинге 6.8 после строки (1) надо добавить следующие строки:

```
dfnumber[v]:= count;
count:= count + 1;
```

Назовем такую нумерацию *глубинной нумерацией* ориентированного графа. Отметим, что эта нумерация обобщает нумерацию, полученную при прямом обходе дерева (см. раздел 3.1).

Всем потомкам вершины v присваиваются глубинные номера, не меньшие, чем номер, присвоенный вершине v. Фактически вершина w будет потомком вершины v тогда и только тогда, когда выполняются неравенства $dfnumber(v) \leq dfnumber(w) \leq dfnumber(v) +$ количество потомков вершины v[1]. Очевидно, что прямые дуги идут от вершин с низкими номерами к вершинам с более высокими номерами, а обратные дуги — от вершин с высокими номерами к вершинам с более низкими номерами.

Все поперечные дуги также идут от вершин с высокими номерами к вершинам с более низкими номерами. Чтобы показать это, предположим, что есть дуга $x \to y$ и выполняется неравенство $dfnumber(x) \leq dfnumber(y)$. Отсюда следует, что вершина x пройдена (посещена) раньше вершины y. Каждая вершина, пройденная в промежуток времени от вызова *dfs(x)* и до завершения *dfs(y),* становится потомком вершины x в глубинном остовном лесу. Если при рассмотрении дуги $x \to y$ вершина y еще не посещалась, то эта дуга становится дугой дерева. В противном случае дуга $x \to y$ будет прямой дугой. Таким образом, если для дуги $x \to y$ выполняется неравенство $dfnumber(x) \leq dfnumber(y)$, то она не может быть поперечной дугой.

В следующих разделах этой главы мы рассмотрим применения метода поиска в глубину для решения различных задач на графах.

---

[1] **Для истинных потомков вершины *v* первое из приведенных неравенств должно быть строгим. — *Прим. ред.***

6.6. Ориентированные ациклические графы

Ориентированный ациклический граф — это орграф, не имеющий циклов. Можно сказать, что ациклический орграф более общая структура, чем дерево, но менее общая по сравнению с обычным ориентированным графом. На рис. 6.18 представлены дерево, ациклический орграф и ориентированный граф с циклом.

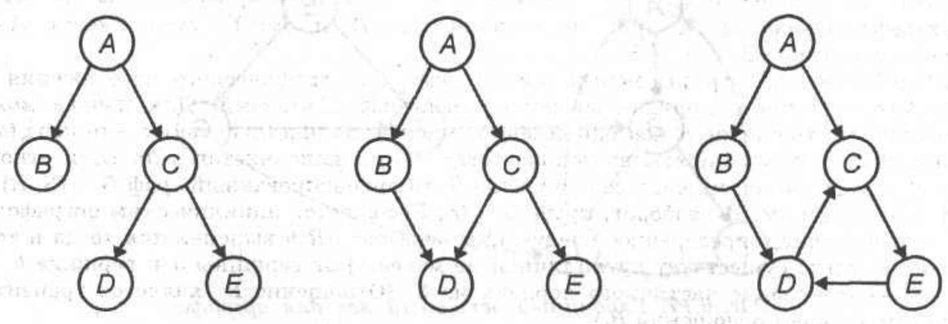


Рис. 6.18. Три ориентированных графа

Ациклические орграфы удобны для представления синтаксических структур арифметических выражений, имеющих повторяющиеся подвыражения. Например, на рис. 6.19 показан ациклический орграф для выражения

$$((a + b)*c + ((a + b) + e) * (e + f)) * ((a + b)*c).$$

Подвыражения $a + b$ и $(a + b)*c$ встречаются в выражении несколько раз, поэтому они представлены вершинами, в которые входят несколько дуг.

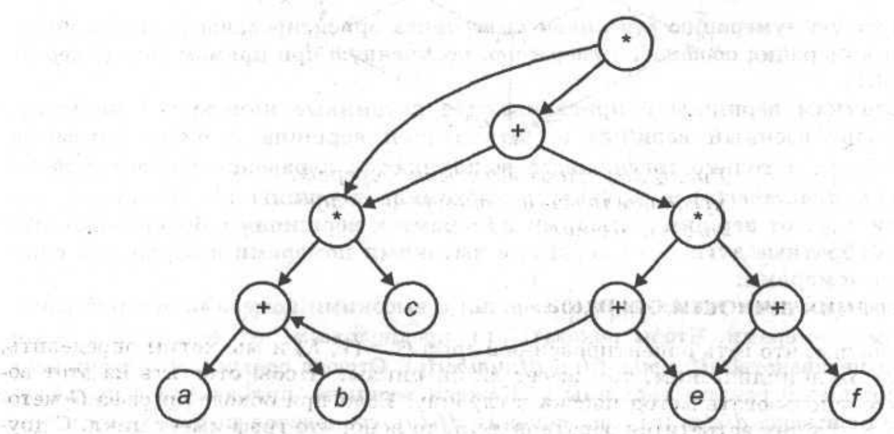


Рис. 6.19. Ориентированный ациклический граф для арифметического выражения

Ациклические орграфы также полезны для представления отношений частичных порядков. Отношением *частичного порядка R*, определенным на множестве *S*, называется бинарное отношение, для которого выполняются следующие условия.

1. Ни для какого элемента a из множества S не выполняется aRa (свойство антирефлексивности).
2. Для любых a, b, c из S, таких, что aRb и bRc, выполняется aRc (свойство транзитивности).

Двумя естественными примерами отношений частичного порядка могут служить отношение "меньше чем" (обозначается знаком "<"), заданное на множестве целых чисел, и отношение строгого включения (⊂) множеств.

Пример 6.14. Пусть S = {1, 2, 3} и P(S) — множество всех подмножеств множества S, т.е. $P(S) = \{\varnothing, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$. Отношение строгого включения (⊂) является отношением частичного порядка на множестве $P(S)$. Очевидно, включение $A \subset A$ не выполняется для любого множества A из P (антирефлексивность), и при выполнении $A \subset B$ и $B \subset C$ выполняется $A \subset C$ (транзитивность). □

Ациклические орграфы можно использовать для графического изображения отношения частичного порядка между какими-либо объектами. Для начала можно представить отношение R как одноименное множество пар (дуг) таких, что пара (a, b) принадлежит этому множеству только тогда, когда выполняется aRb. Если отношение R определено на множестве элементов S, то ориентированный граф G = (S, R) будет ациклическим. И наоборот, пусть G = (S, R) является ациклическим орграфом, а R^+ — отношение, определенное следующим образом: aR^+b выполняется тогда и только тогда, когда существует путь (длиной не менее 1) от вершины a к вершине b. Тогда R^+ — отношение частичного порядка на S. (Отношение R^+ является транзитивным замыканием отношения R.)

Пример 6.15. На рис. 6.20 показан ациклический орграф $(P(S), R)$, где S = {1, 2, 3}. Здесь R^+ — отношение строгого включения на множестве $P(S)$. □

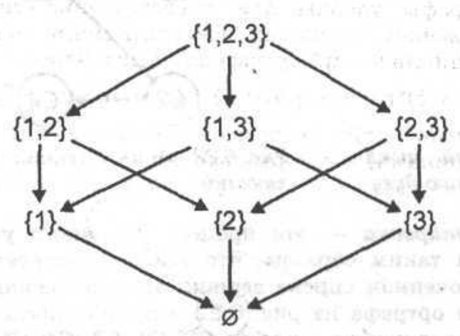


Рис. 6.20. Ациклический орграф, представляющий отношение строгого включения

Проверка ацикличности орграфа

Предположим, что есть ориентированный граф $G = (V, E)$ и мы хотим определить, является ли он ациклическим, т.е. имеет ли он циклы. Чтобы ответить на этот вопрос, можно использовать метод поиска в глубину. Если при обходе орграфа G методом поиска в глубину встретится обратная дуга, то ясно, что граф имеет цикл. С другой стороны, если в орграфе есть цикл, тогда обратная дуга обязательно встретится при обходе этого орграфа методом поиска в глубину.

Чтобы доказать это, предположим, чт9 орграф G имеет цикл. Пусть при обходе данного орграфа методом поиска в глубину вершина v имеет наименьшее глубинное число среди всех вершин, составляющих цикл. Рассмотрим дугу $u \to v$, принадлежащую этому циклу. Поскольку вершина u входит в цикл, то она **должна** быть потомком вершины v в глубинном остовном лесу. Поэтому дуга $u \to v$ не может быть поперечной дугой. Так как глубинный номер вершины u больше глубинного номера вершины v, то отсюда следует, что эта дуга не может быть дугой дерева и прямой дугой. Следовательно, дуга $u \to v$ является обратной дугой, как показано на рис. **6.21**.

Топологическая сортировка

Большие проекты часто разбиваются на совокупность меньших задач, которые выполняются в определенном порядке и обеспечивают полное завершение **целого** проекта. Например, план учебного **заведения** требует определенной последовательности в чтении учебных курсов. Ациклические орграфы могут служить естественной моделью для таких ситуаций. Например, мы создадим дугу от учебного курса **С** к учебному курсу *D* тогда, когда чтение курса С предшествует чтению курса D.

Пример 6.16. На рис. 6.22 показан ациклический орграф, представляющий структуру предшествований пяти учебных курсов. Чтение учебного курса С3, например, требует предварительного чтения курсов С1 и С2. □

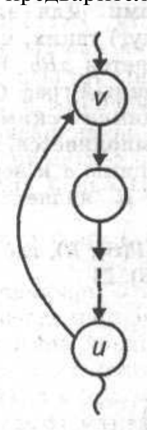


*Рис. **6.21.** Каждый цикл **содержит** обратную дугу*

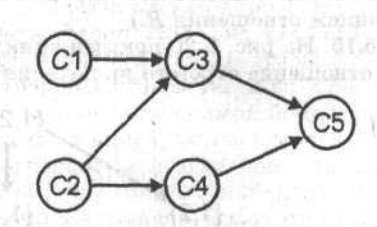


*Рис. 622. Ациклический орграф, **представляющий** структуру предшествований*

Топологическая сортировка — это процесс линейного упорядочивания вершин ациклического орграфа таким образом, что если существует дуга от вершины *i* к вершине *j*, то в упорядоченном списке вершин орграфа вершина i предшествует вершине *j*. Например, для орграфа из рис. 6.22 вершины после топологической сортировки расположатся в следующем порядке: С1, С2, С3, С4, С5.

Топологическую сортировку легко выполнить с помощью модифицированной процедуры листинга 6.8, если в ней после строки (4) добавить оператор вывода на печать:

```
procedure topsort ( v: вершина ) ;
    { печать в обратном топологическом порядке вершин,
        достижимых из вершины v }
  var
        w: вершина;
  begin
        mark[v]:= visited;
        for каждая вершина w из списка L[v] do
            if mark[w] = unvisited then
                topsort(w);
        writeln(v)
  end; { topsort }
```

Когда процедура ***topsort*** заканчивает поиск всех вершин, являющихся потомками вершины ***v***, печатается сама вершина ***v***. Отсюда следует, что процедура ***topsort*** распечатывает все вершины в обратном топологическом порядке.

Эта процедура работает вследствие того, что в ациклическом орграфе нет **обратных** дуг. Рассмотрим, что происходит, когда поиск в глубину достигает конечной

вершины x. Из любой вершины могут исходить только дуги дерева, прямые и поперечные дуги. Но все эти дуги направлены в вершины, которые уже пройдены (посещались до достижения вершины x), и поэтому предшествуют вершине x.

6.7. Сильная связность

Сильно связной компонентой ориентированного графа называется максимальное множество вершин, в котором существуют пути из любой вершины в любую другую вершину. Метод поиска в глубину можно эффективно использовать для нахождения сильно связных компонентов ориентированного графа.

Дадим точную формулировку понятия сильной связности. Пусть $G = (V, E)$ — ориентированный граф. Множество вершин V разбивается на классы эквивалентности V_i, $1 < i < r$, так, что вершины v и w будут эквивалентны тогда и только тогда, когда существуют пути из вершины v в вершину w и из вершины w в вершину v. Пусть E_i, $1 < i < r$, — множество дуг, которые начинаются и заканчиваются в множестве вершин V_i. Тогда графы $G_i = (V_i, E_i)$ называются *сильно связными компонентами* графа G. Ориентированный граф, состоящий только из одной сильно связной компоненты, называется *сильно связным*.

Пример **6.17**. На рис. 6.23 представлен ориентированный граф с двумя сильно связными компонентами, которые показаны на рис. 6.24. D

Отметим, что каждая вершина ориентированного графа *G* принадлежит какой-либо сильно связной компоненте, но некоторые дуги могут не принадлежать никакой сильно связной компоненте. Такие дуги идут от вершины одной компоненты к вершине, принадлежащей другой компоненте. Можно представить связи между компонентами путем создания *редуцированного (приведенного) графа* для графа *G*. Вершинами приведенного графа являются сильно связные компоненты графа *G*. В приведенном графе дуга от вершины *C* к вершине *D* существует только тогда, когда в графе *G* есть дуга от какой-либо вершины, принадлежащей компоненте *C*, к какой-либо вершине, принадлежащей компоненте D. Редуцированный граф всегда является ациклическим орграфом, поскольку если бы существовал цикл, то все компоненты, входящие в этот цикл, в действительности были бы одной связной компонентой. На рис. 6.25 показан редуцированный граф для графа из рис. 6.23.

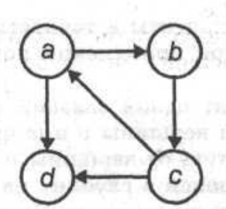


Рис. 6.23. Ориентированный граф

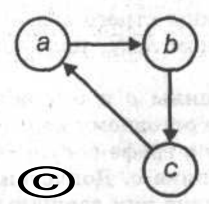


Рис. 624. Сильно связные компоненты орграфа из рис. 6.23

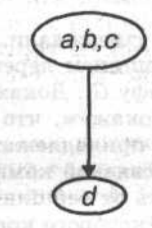


Рис. 6.25. Редуцированный граф

Теперь рассмотрим алгоритм нахождения сильно связных компонент для заданного ориентированного графа G.

1. Сначала выполняется поиск в глубину на графе G. Вершины нумеруются в порядке завершения рекурсивно вызванной процедуры *dfs*, т.е. присвоение номеров вершинам происходит после строки (4) в листинге 6.8.
2. Конструируется новый ориентированный граф G_r путем обращения направления всех дуг графа G.

3. Выполняется поиск в глубину на графе G_r, начиная с вершины с наибольшим номером, присвоенным на шаге 1. Если проведенный поиск не охватывает всех вершин, то начинается новый поиск с вершины, имеющей наибольший номер среди оставшихся **вершин**.

4. Каждое дерево в полученном остовном лесу является сильно связной компонентой графа *G*.

Пример 6.18. Применим описанный алгоритм к ориентированному графу, представленному на рис. 6.23. Прохождение вершин графа начнем с вершины а и затем перейдем на вершину *Ъ*. Присвоенные после завершения шага 1 номера вершин показаны на рис. 6.26. Полученный путем обращения дуг граф G_r представлен на рис. 6.27.

Выполнив поиск в глубину на графе G_r, получим глубинный остовный лес, показанный на рис. 6.28. Обход остовного леса мы начинаем с вершины а, принимаемой в качестве корня дерева, так как она имеет самый высокий номер. Из корня *а* можно достигнуть только вершины *с* и *Ь*. Следующее дерево имеет только корень *d*, поскольку вершина *d* имеет самый высокий номер среди оставшихся (но она и единственная оставшаяся вершина). Каждое из этих деревьев формирует отдельную сильно связанную компоненту исходного графа *G*. □

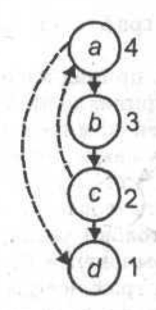


Рис. 6.26. Номера вершин после выполнения шага 1 алгоритма

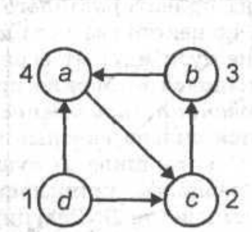


Рис. 6.27. Граф G_r

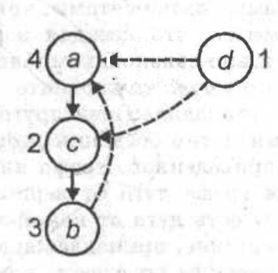


Рис. 6.28. Глубинный остовный лес

Выше мы утверждали, что вершины строго связной компоненты в точности соответствуют вершинам дерева в остовном лесу, полученном при применении поиска в глубину к графу G_r. Докажем это.

Сначала покажем, что если вершины *v* и *w* принадлежат одной связной компоненте, то они принадлежат и одному остовному дереву. Если вершины *v* и *w* принадлежат одной связной компоненте, то в графе G существует путь от вершины *v* к вершине *w* и путь от вершины *w* к вершине *v*. Допустим, что поиск в глубину на графе G_r начат от некоторого корня *x* и достиг или вершины *v*, или вершины *w*. Поскольку существуют пути (в обе стороны) между вершинами *v* и *w*, то обе они принадлежат остовному дереву с корнем *x*.

Теперь предположим, что вершины *v* и *w* принадлежат одному и тому же остовному дереву в глубинном остовном лесу графа *G*. Покажем, что они принадлежат одной и той же сильно связной компоненте. Пусть *x* — корень остовного дерева, которому принадлежат вершины *v* и *w*. Поскольку вершина *v* является потомком вершины *x*, то в графе G_r существует путь от вершины *x* к вершине *v*, а в графе *G* — путь от вершины *v* к вершине *x*.

В соответствии с прохождением вершин графа *G*, методом поиска в глубину вершина *v* посещается позднее, чем вершина *x*, т.е. вершина *x* имеет более высокий номер, чем вершина *v*. Поэтому при обходе графа *G* рекурсивный вызов процедуры *dfs* для вершины *v* заканчивается раньше, чем вызов *dfs* для вершины *x*. Но если при обходе графа G вызывается процедура *dfs* для вершины *v*, то из-за наличия пути от

вершины *v* к вершине *x* процедура *dfs* для вершины *x* начнется и закончится до окончания процедуры *dfs(v)*, и, следовательно, вершина *x* получит меньший номер, чем вершина *v*. Получаем противоречие.

Отсюда заключаем, что при обходе графа *G* вершина *v* посещается при выполнении *dfs(x)* и поэтому вершина *v* является потомком вершины *x*. Следовательно, в графе G существует путь от вершины *x* к вершине *v*. Более того, так как существует и путь от вершины *v* к вершине *x*, то вершины *x* и *v* принадлежат одной сильно связной компоненте. Аналогично доказывается, что вершины *x* и *w* принадлежат одной сильно связной компоненте. Отсюда вытекает, что и вершины *v* и w принадлежат той же сильно связной компоненте, так как существует путь от вершины *v* к вершине *w* через вершину *x* и путь от вершины *w* к вершине *v* через вершину *x*.

Упражнения

6.1. Представьте граф, изображенный на рис. 6.29, посредством

а) матрицы смежности, содержащей стоимости дуг;

б) связанных списков смежности, показывающих стоимость дуг.

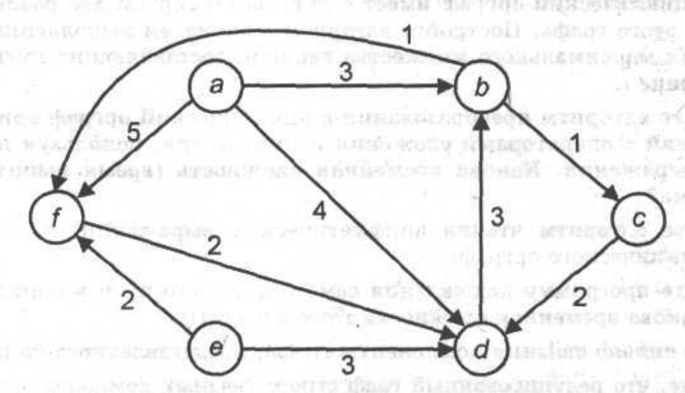


Рис. 6.29. Ориентированный граф с помеченными дугами

6.2. Постройте математическую модель для следующей задачи составления календарного плана работ. Есть множество задач T_1, T_2, ..., T_n, для выполнения которых необходимо время t_1, t_2, ..., t_n, и множество отношений вида "задача T_i должна закончиться до начала задачи T_j". Найдите минимальное время, необходимое для выполнения всех задач.

6.3. Выполните реализацию операторов FIRST, NEXT и VERTEX для орграфов, представленных посредством

а) матриц смежности;

б) связанных списков смежности;

в) списков смежности, показанных на рис. 6.5.

6.4. Для графа, показанного на рис. 6.29, используйте

а) алгоритм Дейкстры для нахождения кратчайших путей от вершины а до остальных вершин;

б) алгоритм Флойда для нахождения кратчайших путей между всеми парами вершин.

6.5. Напишите законченную программу алгоритма Дейкстры, используя связанные списки смежности и очередь с приоритетами, реализованную посредством частично упорядоченного дерева.

*6.6. Покажите, **что** алгоритм Дейкстры работает неправильно, если стоимости дуг могут быть отрицательными.

**6.7. Покажите, что алгоритм Флойда работает корректно, если некоторые дуги имеют отрицательную стоимость, но нет циклов, состоящих из дуг с отрицательными стоимостями.

6.8. Полагая, что вершины графа из рис. 6.29 упорядочены как *a, Ъ, ..., /*, постройте **глубинный** остовный лес. Укажите дуги дерева, прямые, обратные и поперечные, а также подсчитайте глубинные номера вершин.

*6.9. Пусть есть глубинный остовный лес и **совершается** обход составляющих его **остовных** деревьев слева направо в обратном порядке. Покажите, что в этом случае список вершин будет совпадать со списком вершин, полученных с помощью процедуры *dfs,* вызываемой при построении **остовного** леса.

6.10. *Корнем* ациклического орграфа называется вершина *r* такая, что существуют пути, исходящие из этой вершины и достигающие всех остальных вершин орграфа. Напишите программу, определяющую, имеет ли данный ациклический орграф корень.

*6.11. Пусть ациклический орграф имеет *e* дуг. Зафиксируем две различные вершины *s* и *t* этого графа. Постройте алгоритм с временем выполнения *O(e)* для нахождения максимального множества вершин, составляющих пути от вершины s к вершине *t*.

6.12. Постройте алгоритм преобразования в ациклический орграф арифметических выражений с операторами сложения и умножения, используя повторяющиеся подвыражения. Какова временная сложность (время выполнения) этого алгоритма?

6.13. Постройте алгоритм **чтения арифметических** выражений, представленных в виде ациклического орграфа.

6.14. Напишите программу **нахождения** самого длинного пути в ациклическом орграфе. Какова временная сложность этого алгоритма?

6.15. Найдите сильно связные компоненты графа, представленного на рис. 6.29.

*6.16. Докажите, что редуцированный граф строго связных компонент (см. раздел 6.7) обязательно должен быть ациклическим орграфом.

6.17. Нарисуйте первый остовный лес, обратный граф и второй остовный лес в процессе определения сильно связных компонент орграфа, показанного на рис. 6.29.

6.18. Реализуйте алгоритм нахождения сильно связных компонент, рассмотренный в разделе 6.7.

*6.19. Покажите, что алгоритм нахождения сильно связных компонент требует времени порядка *O(e)* на ориентированном графе, имеющем *e* дуг и *n* вершин, если $n < e$.

*6.20. Напишите программу, на входе которой задается орграф и две его вершины. Программа должна распечатывать все простые пути, ведущие от одной вершины к другой. Какова временная сложность (время выполнения) этой программы?

*6.21. *Транзитивная редукция* ориентированного графа $G = (V, E)$ определяется как произвольный граф $G' = (V, E')$, имеющий то же множество вершин, но с минимально возможным числом дуг, транзитивное замыкание которого совпадает с транзитивным замыканием графа *G*. Покажите, что если граф G ацикличен, то его транзитивная редукция единственна.

*6.22. Напишите программу нахождения транзитивной редукции орграфа. Какова временная сложность этой программы?

6.23.** Орграф $G' = (V, E')$ называется ***минимальным эквивалентным орграфом для орграфа $G = (V, E)$, если E' — наименьшее подмножество множества E такое, что транзитивные замыкания обоих орграфов G и G' совпадают. Покажите, что если граф G ацикличен, то для него существует только один минимальный эквивалентный орграф, а именно — его транзитивная редукция.

***6.24.** Напишите программу нахождения минимального эквивалентного орграфа. Какова временная сложность этой программы?

***6.25.** Напишите программу нахождения самого длинного простого пути от заданной вершины орграфа. Какова временная сложность этой программы?

Библиографические примечания

Хорошими источниками дополнительных сведений по теории графов могут служить монографии [11] и [47][1]. В книгах [24], [28] и [110] рассматриваются различные алгоритмы на графах.

Алгоритм нахождения кратчайших путей от одного источника, описанный в разделе 6.3, взят из работы **Дейкстры** [26]. Алгоритм нахождения всех кратчайших путей описан Флойдом [33], а алгоритм транзитивного замыкания — Уоршеллом [114]. В работе [57] приведен эффективный алгоритм нахождения кратчайших путей в разреженных графах. В книге [63] можно найти дополнительный материал по топологической сортировке.

Алгоритм определения сильно связных компонент, описанный в разделе 6.7, подобен алгоритму, предложенному Косерейю (R. **Kosaraju**) в 1978 г. (неопубликовано), и алгоритму, опубликованному в статье [99]. Тарьян **(Tarjan)** [107] предложил другой алгоритм определения сильно связных компонент, который требует только одного обхода графа методом поиска в глубину.

В [19] приведено много примеров использования графов для решения задач планирования и расписаний, подобных упражнению 6.2. В работе [2] доказана единственность транзитивной редукции для ациклического орграфа и что нахождение транзитивной редукции орграфа вычислительно эквивалентно нахождению транзитивного замыкания (упражнения 6.21 и 6.22). С другой стороны, нахождение минимального эквивалентного орграфа (упражнения 6.23 и 6.24) является вычислительно более сложной задачей — это **NP-полная** задача [93].

[1] Обе книги переведены на русский язык (см. библиографию в конце книги). — *Прим. ред.*

ГЛАВА 7

Неориентированные графы

Неориентированный граф $G = (V, E)$ состоит из конечного множества вершин V и множества ребер E. В отличие от ориентированного графа, здесь каждое ребро (v, w) соответствует *неупорядоченной* паре вершин[1]: если (v, w) — неориентированное ребро, то $(v, w) = (w, v)$. Далее неориентированный граф мы будем называть просто графом.

Графы широко используются в различных областях науки и техники для моделирования симметричных отношении между объектами. Объекты соответствуют вершинам графа, а ребра — отношениям между объектами. В этой главе будут описаны различные структуры данных, которые применимы для представления графов. Мы также рассмотрим алгоритмы решения трех типовых задач, возникающих при работе с графами: построения минимальных остовных деревьев, определения двусвязных компонент и нахождения максимального паросочетания графа.

7.1. Основные определения

Многое из терминологии ориентированных графов применимо к неориентированным графам. Например, вершины v и w называются *смежными,* если существует ребро (v, w). Мы будем также говорить, что ребро (v, w) *инцидентно* вершинам v и w.

Путем называется такая последовательность вершин $v_1, v_2 \ldots v_n$, что для всех i, $1 < i < n$, существуют ребра (v_i, v_{i+1}). Путь называется *простым,* если все вершины пути различны, за исключением, возможно, вершин v_1 и v_n. Длина пути равна количеству ребер, составляющих путь, т.е. длина равна $n - 1$ для пути из n вершин. Если для вершин v_1 и v_n существует путь $v_1, v_2 \ldots v_n$, то эти вершины называются *связанными.* Граф называется *связным,* если в нем любая пара вершин связанная.

Пусть есть граф $G = (V, E)$ с множеством вершин V и множеством ребер E. Граф $G' = (V', E')$ называется подграфом графа G, если

1. множество V' является подмножеством множества V,
2. множество E' состоит из ребер (v, w) множества E таких, что обе вершины v и w принадлежат V'.

Если множество E' состоит из *всех* ребер (v, w) множества E таких, что обе вершины v и w принадлежат V', то в этом случае граф G' называется *индуцированным подграфом* графа G.

Пример 7.1. На рис. 7.1,а показан граф $G = (V, E)$ с множеством вершин $V = \{a, b, c, d\}$ и множеством дуг $E = \{(a, b), (a, d), (b, c), (b, d), (c, d)\}$. На рис. 7.1,б представлен один из его индуцированных подграфов, заданный множеством вершин $V' = \{a, b, c\}$ и содержащий все ребра, не инцидентные вершине d. □

Связной компонентой графа G называется максимальный связный индуцированный подграф графа G.

Пример 7.2. Граф, показанный на рис. 7.1,а, является связным графом и имеет только одну связную компоненту, а именно — самого себя. На рис. 7.2 представлен несвязный граф, состоящий из двух связных компонент. □

[1] Пока не будет сказано другое, будем считать, что ребро всегда соответствует паре различных вершин.

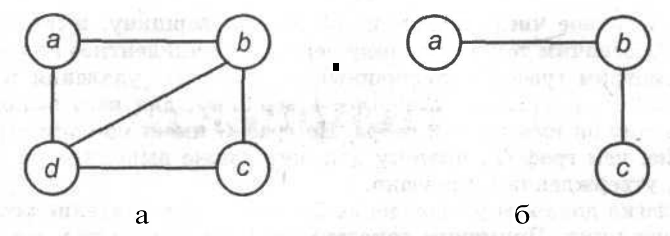


Рис. 7.1. Граф и один их его подграфов

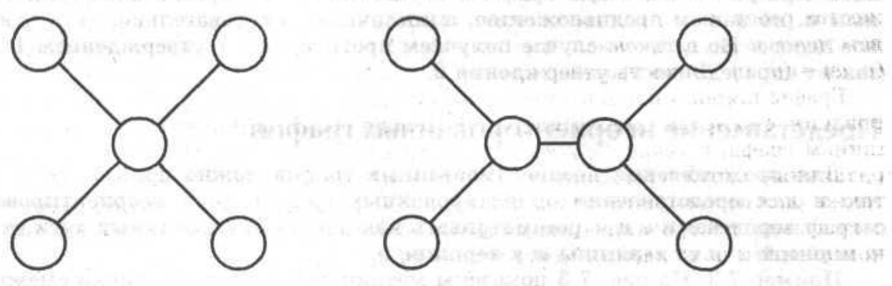


Рис. 7.2. Несвязный граф

Циклом (простым) называется путь (простой) длины не менее 3 от какой-либо вершины до нее самой. Мы не считаем циклами пути длиной 0, длиной 1 (петля от вершины v к ней самой) и длиной 2 (путь вида v, w, v). Граф называется *циклическим,* если имеет хотя бы один цикл. Связный ациклический граф, представляющий собой "дерево без корня", называют *свободным деревом.* На рис. 7.2 показан граф, состоящий из двух связных компонент, каждая из которых является свободным деревом. Свободное дерево можно сделать "обычным" деревом, если какую-либо вершину назначить корнем, а все ребра сориентировать в направление от этого корня.

Свободные деревья имеют два важных свойства, которые мы будем использовать в следующих разделах.

1. Каждое свободное дерево с числом вершин n, $n > 1$, имеет в точности $n - 1$ ребер.
2. Если в свободное дерево добавить новое ребро, то обязательно получится цикл.

Мы докажем первое утверждение методом индукции по n. Очевидно, что утверждение справедливо для $n = 1$, поскольку в этом случае имеем только одну вершину и не имеем ребер. Пусть утверждение 1 справедливо для свободного дерева с $n - 1$ вершинами. Рассмотрим дерево G с n вершинами.

Сначала покажем, что в свободном дереве существуют вершины, имеющие одно инцидентное ребро. Заметим, что G не содержит изолированных вершин (т.е. вершин, не имеющих инцидентных ребер), иначе граф G не был бы связным. Теперь создадим путь от некоторой вершины v_1, следуя по произвольному ребру, инцидентному вершине v_1. На каждом шаге построения этого пути, достигнув какой-либо вершины, выбираем инцидентное ей ребро, которое ведет к вершине, еще не участвовавшей в формировании пути. Пусть таким способом построен путь $v_1, v_2, \dots, v_k$. Вершина v_k будет смежной либо с одной вершиной v_{k-1}, либо еще с какой-нибудь, не входящей в построенный путь (иначе получится цикл). В первом случае получаем, что вершина v_k имеет только одно инцидентное ей ребро, и значит, наше утверждение о том, что в свободном дереве существуют вершины, имеющие одно инцидентное ребро, будет доказано. Во втором случае обозначим через v_{k+1} вершину, смежную с вершиной v_k, и строим путь $v_1, v_2, \dots, v_k, v_{k+1}$. В этом пути все вершины различны (иначе опять получится цикл). Так как количество вершин конечно, то этот процесс

закончится за конечное число шагом и мы найдем вершину, имеющую одно инцидентное ребро. Обозначим такую вершину через **v**, а инцидентное ей ребро — **(v, w)**.

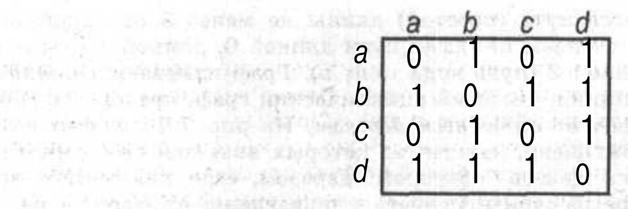
Теперь рассмотрим граф **G'**, полученный в результате удаления из графа G вершины v и ребра (v. w). Граф **G'** имеет **n** - 1 вершину, для него выполняется утверждение 1 и поэтому он имеет **n** - *2* ребра. Но граф G имеет на одну вершину и на одно ребро больше, чем граф **G'**, поэтому для него также выполняется утверждение 1. Следовательно, утверждение 1 доказано.

Теперь мы легко докажем утверждение 2 о том, что добавление ребра в свободное дерево формирует цикл. Применим доказательство от противного, т.е. предположим, что добавление ребра в свободное дерево не формирует цикл. Итак, после добавления нового ребра мы получили граф с n вершинами и n ребрами. Этот граф остался связным и, по нашем предположению, ацикличным. Следовательно, этот граф — свободное дерево. Но в таком случае получаем противоречие с **утверждением 1.** Отсюда вытекает справедливость утверждения 2.

Представление неориентированных графов

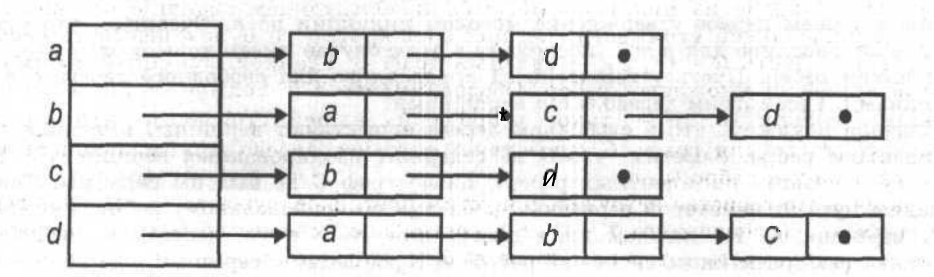
Для представления неориентированных графов можно применять те же методы, что и для представления ориентированных графов, если **неориентированное** ребро между вершинами *v и w* рассматривать как две ориентированных дуги от вершины v *к* вершине ш и от вершины *w к* вершине **v**.

Пример 7.3. На рис. 7.3 показаны матрица смежности и списки смежности, представляющие граф из рис. **7.1,а.** D



	a	b	c	d
a	0	1	0	1
b	1	0	1	1
c	0	1	0	1
d	1	1	1	0

а. Матрица смежности



б. Списки смежности

Рис. 7.3. Представления неориентированного графа

Очевидно, что матрица смежности для неориентированного графа симметрична. Отметим, что в случае представления графа посредством списков смежности для существующего ребра *(i, j)* в списке смежности вершины *i* присутствует вершина ***j***, а в списке смежности вершины *j* — вершина *i*.

7.2. Остовные деревья минимальной стоимости

Пусть G = (V, E) — связный граф, в котором каждое ребро (v, w) помечено числом $c(v, w)$, которое называется *стоимостью ребра*. ***Остовным деревом*** графа G называется свободное дерево, содержащее все вершины V графа G. ***Стоимость*** остовного дерева вычисляется как сумма стоимостей всех ребер, входящих в это дерево. В этом разделе мы рассмотрим методы нахождения остовных деревьев минимальной стоимости.

Пример 7.4. На рис. 7.4 показаны граф с помеченными ребрами и его остовное дерево минимальной стоимости. О

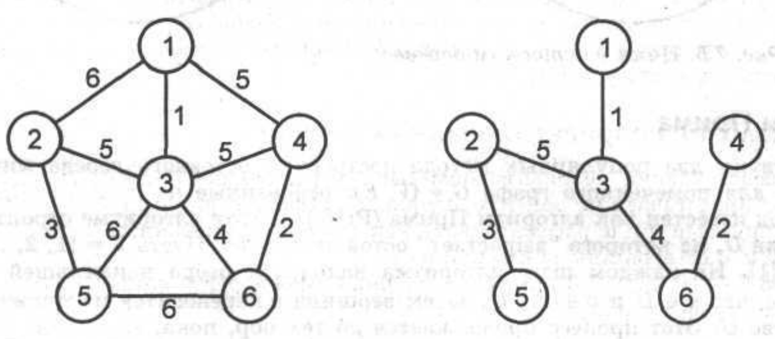


Рис. 7.4. Граф и его остовное дерево

Типичное применение остовных деревьев минимальной стоимости можно найти при разработке коммуникационных сетей. Здесь вершины графа представляют города, ребра — возможные коммуникационные линии между городами, а стоимость ребер соответствует стоимости коммуникационных линий. В этом случае остовное дерево минимальной стоимости представляет коммуникационную сеть, объединяющую все города коммуникационными линиями минимальной стоимости.

Свойство остовных деревьев минимальной стоимости

Существуют различные методы построения остовных деревьев минимальной стоимости. Многие из них основываются на следующем свойстве остовных деревьев минимальной стоимости, которое для краткости будем называть *свойством ОДМС.* Пусть G = (V, E) — связный граф с заданной функцией стоимости, определенной на множестве ребер. Обозначим через U подмножество множества вершин V. Если (u, v) — такое ребро наименьшей стоимости, что u е U и v е V \ U, тогда для графа G существует остовное дерево минимальной стоимости, содержащее ребро (и, v).

Доказать это свойство нетрудно. Допустим противное: существует остовное дерево для графа G, обозначим его T — (V, E'), содержащее множество U и не содержащее ребро (u, v), стоимость которого меньше любого остовного дерева для G, содержащего ребро (u, v).

Поскольку дерево T — свободное дерево, то из второго свойства свободных деревьев следует, что добавление ребра (u, v) *к* этому дереву приведет к образованию цикла. Этот цикл содержит ребро (и, v) и будет содержать другое ребро (u', v') такое, что u € U и v е $V \setminus U$, как показано на рис. 7.5. Удаление ребра (u, v) приведет к разрыву цикла и образованию остовного дерева, чья стоимость будет не выше стоимости дерева T, поскольку $c(u, v) < c(u', v)$. Мы пришли к противоречию с предположением, что остовное дерево T — это дерево минимальной стоимости.

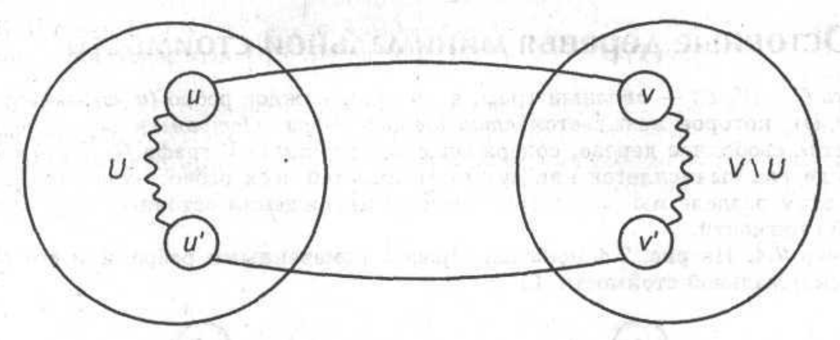


Рис. 7.5. Цикл в остовном дереве

Алгоритм Прима

Существуют два популярных метода построения остовного дерева минимальной стоимости для помеченного графа $G = (V, E)$, основанные на свойстве ОДМС. Один такой метод известен как алгоритм Прима (Prim). В этом алгоритме строится множество вершин U, из которого "вырастает" остовное дерево. Пусть $V = \{1, 2, ..., n\}$. Сначала $U = \{1\}$. На каждом шаге алгоритма находится ребро наименьшей стоимости (и, v) такое, что $u \in U$ и $v \in V \setminus U$, затем вершина v переносится из множества $V \setminus U$ в множество U. Этот процесс продолжается до тех пор, пока множество U не станет равным множеству V. Эскиз алгоритма показан в листинге 7.1, а процесс построения остовного дерева для графа из рис. 7.4,а — на рис. 7.6.

Листинг 7.1. Алгоритм Прима

```
procedure Prim ( G: граф; var T: множество ребер );
    var
        U: множество вершин;
        u, v. вершина;
    begin
        T:= 0;
        U:= {1};
        while U ≠ V do begin
            нахождение ребра (u, v) наименьшей стоимости и такого,
                что u £ У и v ∈ V\U;
            T:= Г u {(u, v)};
            U:= U U {v}
        end
    end; { Prim }
```

Если ввести два массива, то можно сравнительно просто организовать на каждом шаге алгоритма выбор ребра с наименьшей стоимостью, соединяющего множества U и $V \setminus U$. Массив *CLOSEST*[*i*] для каждой вершины i из множества $V \setminus U$ содержит вершину из U, с которой он соединен ребром минимальной стоимости (это ребро выбирается среди ребер, инцидентных вершине i, и которые ведут в множество U). Другой массив *LOWCOST*[*i*] хранит значение стоимости ребра (*i*, *CLOSEST*[*i*]).

На каждом шаге алгоритма просматривается массив *LOWCOST*, находится минимальное значение *LOWCOST*[*k*]. Вершина k принадлежит множеству $V \setminus U$ и соединена ребром с вершиной из множества U. Затем выводится на печать ребро (*k*, *CLOSEST*[*k*]). Так как вершина k присоединяется к множеству U, то вследствие этого изменяются массивы *LOWCOST* и *CLOSEST*. Программа на языке Pascal, реализующая алгоритм Прима, представлена в листинге 7.2. На вход программы посту-

пает массив С размера *n* x *n*, чьи элементы *C[i, j]* равны стоимости ребер (i, *j*). Если ребра (i, *j*) не существуют, то элемент *C[i,j]* полагается равным некоторому достаточно большому числу.

После нахождения очередной вершины *k* остовного дерева *LOWCOST[k]* ложится равным *infinity* (бесконечность), очень большому числу, такому, чтобы эта вершина уже в дальнейшем не рассматривалась. Значение числа *infinity* должно быть больше стоимости любого ребра графа.

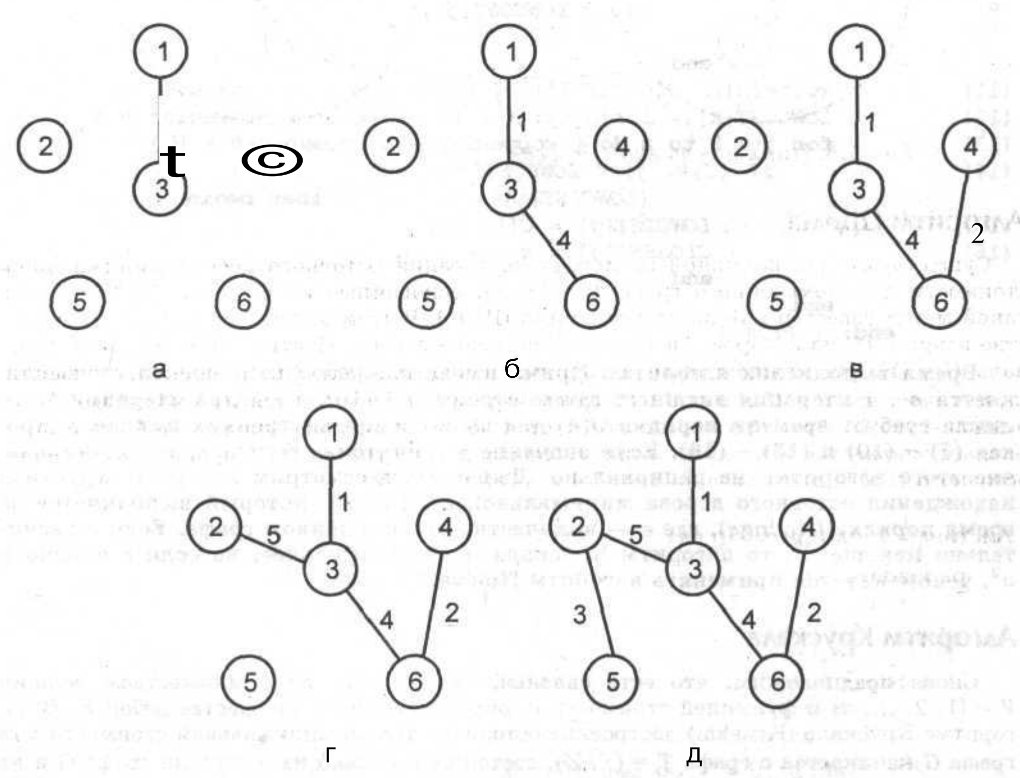


Рис. 7.6. Последовательность построения остовного дерева алгоритмом Прима

Листинг 7.2. Программа выполнения алгоритма Прима

```
    procedure Prim ( C: array[1..n, 1..n] of real );
        { Prim печатает ребра остовного дерева минимальной стоимости
            для графа с вершинами {1, ..., n} и матрицей стоимости C}
        var
            LOWCOST: array[1..n] of real;
            CLOSEST: array[1..n] of integer;
            i, j, k, min: integer;
            { i и j — индексы. При просмотре массива LOWCOST
                k — номер найденной вершины, rain = LOWCOST[k] }
        begin
(1)         for i:= 2 to n do begin
                { первоначально в множестве U только вершина 1 }
(2)             LOWCOST[i] := C[l, i] ;
(3)             CLOSEST[i]-.=1
            end;
```

```
(4)         for i:= 2 to n do begin
                  { поиск вне множества U наилучшей вершины k, имеющей
                        инцидентное ребро, оканчивающееся в множестве U }
(5)               min:= LOWCOST[2];
(6)               k:= 2;
(7)               for j:= 3 to n do
(8)                     if LOWCOST[j] < min then begin
(9)                           min:= LOWCOST[j];
(10)                          k:= j
                        end;
(11)        writeln(k, CLOSEST[k]); { вывод ребра на печать }
(12)        LOWCOST[k]:= infinity; { k добавляется в множество U }
(13)        for j:= 2 to n do { корректировка стоимостей в U }
(14)              if (C[k, j] < LOWCOST[j]) and
                        (LOWCOST[j] < infinity) then begin
(15)                    LOWCOST[j]:= C[k, j];
(16)                    CLOSEST[j] := k
                        end
            end
      end; { Prim }
```

Время выполнения алгоритма Прима имеет порядок $O(n^2)$, поскольку выполняется n — 1 итерация внешнего цикла строк (4) — (16), а каждая итерация этого цикла требует времени порядка $O(n)$ для выполнения внутренних циклов в строках (7) - (10) и (13) - (16). Если значение га достаточно большое, то использование этого алгоритма не рационально. Далее мы рассмотрим алгоритм Крускала нахождения остовного дерева минимальной стоимости, который выполняется за время порядка $O(e \log e)$, где e — количество ребер в данном графе. Если e значительно меньше иг, то алгоритм Крускала предпочтительнее, но если e близко к n^2, рекомендуется применять алгоритм Прима.

Алгоритм Крускала

Снова предположим, что есть связный граф $G = (V, E)$ с множеством вершин $V = \{1, 2, ..., n\}$ и функцией стоимости с, определенной на множестве ребер E. В алгоритме Крускала (Kruskal) построение остовного дерева минимальной стоимости для графа G начинается с графа $T = (V, 0)$, состоящего только из га вершин графа G и не имеющего ребер. Таким образом, каждая вершина является связной (с самой собой) компонентой. В процессе выполнения алгоритма мы имеем набор связных компонент, постепенно объединяя которые формируем остовное дерево.

При построении связных, постепенно возрастающих компонент поочередно проверяются ребра из множества E в порядке возрастания их стоимости. Если очередное ребро связывает две вершины из разных компонент, тогда оно добавляется в граф T. Если это ребро связывает две вершины из одной компоненты, то оно отбрасывается, так как его добавление в связную компоненту, являющуюся свободным деревом, приведет к образованию цикла. Когда все вершины графа G будут принадлежать одной компоненте, построение остовного дерева минимальной стоимости T для этого графа заканчивается.

Пример 7.5. Рассмотрим помеченный граф, представленный на рис. 7.4,а. Последовательность добавления ребер в формирующееся дерево T показана на рис. 7.7. Ребра стоимостью 1, 2, 3 и 4 рассмотрены первыми и все включены в T, поскольку их добавление не приводит к циклам. Ребра (1, 4) и (3, 4) стоимостью 5 нельзя включить в T, так как они соединяют вершины одной и той же компоненты (рис. 7.7,г) и поэтому замыкают цикл. Но оставшееся ребро (2, 3) также стоимостью 5 не создает цикл. После включения этого ребра в T формирование остовного дерева завершается. D

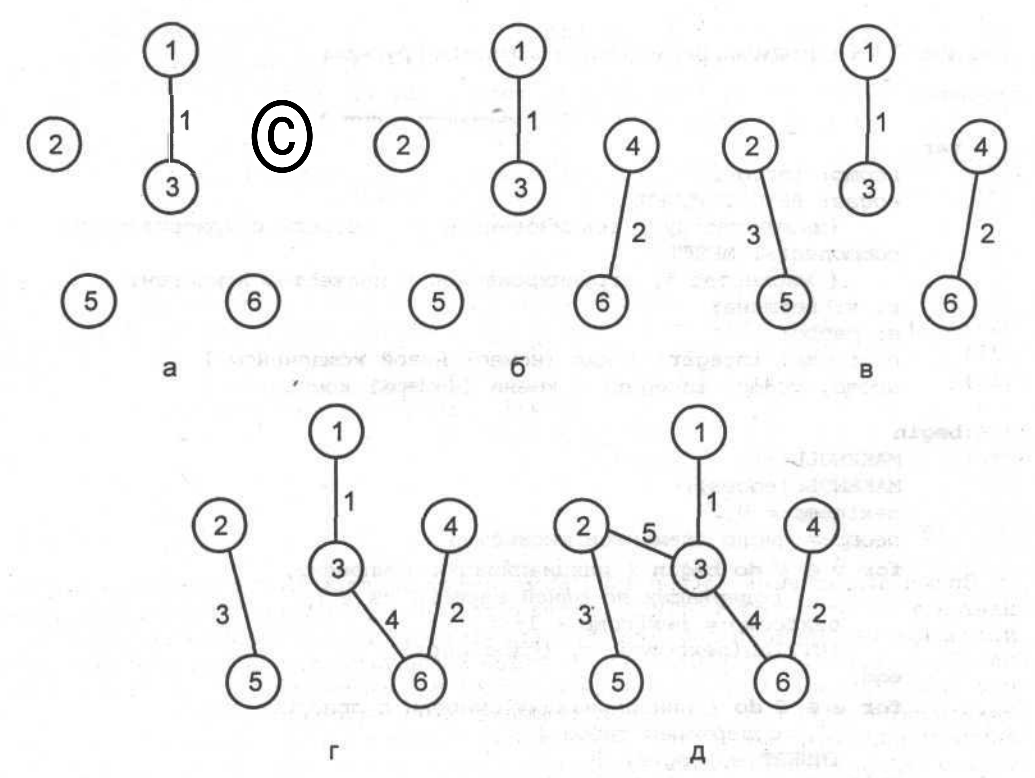


Рис. 7.7. Последовательное формирование остовного дерева минимальной стоимости посредством алгоритма Крускала

Этот алгоритм можно реализовать, основываясь на множествах (для вершин и ребер) и операторах, рассмотренных в главах 4 и 5. Прежде всего, необходимо множество ребер *E*, к которому можно было бы последовательно применять оператор DELETEMIN для отбора ребер в порядке возрастания их стоимости. Поэтому множество ребер целесообразно представить в виде очереди с приоритетами и использовать для нее частично упорядоченное дерево в качестве структуры данных.

Необходимо также поддерживать набор *C* связных компонент, для чего можно использовать следующие операторы.

1. Оператор **MERGE(*A*, *B*, *C*)** объединяет компоненты *A* и *B* из набора *C* и результат объединения помещает или в A, или в *B*.[1]
2. Функция **FIND(*v*, *C*)** возвращает имя той компоненты из набора *C*, которая содержит вершину *v*.
3. Оператор **INITIAL(*A*, *v*, C)** создает в наборе C новую компоненту с именем *A*, содержащую только одну вершину *v*.

Напомним, что в разделе 5.5 для множеств, поддерживающих операторы MERGE и FIND, мы ввели абстрактный тип данных, который назвали MFSET. В эскизе программы (листинг 7.3), реализующей алгоритм Крускала, **используется** этот тип данных.

---

[1] Отметим, что операторы MERGE и **FIND** здесь несколько отличаются от одноименных операторов из раздела 5.5, поскольку сейчас C — параметр, указывающий, где находятся множества *A* к *B*.

Листинг 7.3. Программа, реализующая алгоритм Крускала

```
procedure Kruskal ( V: SET; E: SET; var T: SET ) ;
    { V — множество вершин, £ и Г — множество дуг }
    var
        ncomp: integer; { текущее количество компонент }
        edges: PRIORITYQUEUE;
            {множество дуг, реализованное как очередь с приоритетами}
        components: MFSET;
            { множество V, сгруппированное в множество компонент }
        u, V. вершина;
        e: ребро;
        nextcomp: integer; { имя (номер) новой компоненты }
        ucomp, vcomp: integer; { имена (номера) компонент }

    begin
        MAKENULL(T) ;
        MAKENULL(edges) ;
        nextcomp:= 0;
        ncomp:= число элементов множества V;
        for v e V do begin { инициализация компонент,
                содержащих по одной вершине из V }
            nextcomp:= nextcomp + 1;
            INITIAL(nextcomp, v, components)
        end;
        for e e E do { инициализация очереди с приоритетами,
                содержащей ребра }
            INSERT(e, edges);
        while ncomp > 1 do begin { рассматривается следующее ребро }
            e:= DELETEMIN(edges);
            пусть e = (u, v);
            ucomp:= FIND(u, components);
            vcomp:= FIND(v, components);
            if ucomp <> vcomp then begin
                { ребро e соединяет две различные компоненты }
                MERGE(ucomp, vcomp, components);
                ncomp:= ncomp - 1;
                INSERT(e, T)
            end
        end
    end; { Kruskal }
```

Для реализации операторов, используемых в этой программе, можно применить методы, описанные в разделе 5.5. Время выполнения этой программы зависит от двух факторов. Если в исходном графе G всего e ребер, то для вставки их в очередь с приоритетами потребуется время порядка $O(e \log e)$.[1] Каждая итерация цикла **while** для нахождения ребра с наименьшей стоимостью в очереди *edges* требует времени порядка $O(\log e)$. Поэтому выполнение всего этого цикла в самом худшем случае потребует времени $O(e \log e)$. Вторым фактором, влияющим на время выполнения программы, является общее время выполнения операторов MERGE и FIND, которое зависит от метода реализации АТД MFSET. Как показано в разделе 5.5, существуют методы, требующие времени и $O(e \log e)$, и $O(e\,\alpha(e))$. В любом случае алгоритм Крускала может быть выполнен за время $O(e \log e)$.

---

[1] **Можно заранее создать частично упорядоченное дерево с e элементами за время $O(e)$. Мы рассмотрим этот прием в разделе 8.4 главы 8.**

7.3. Обход неориентированных графов

Во многих задачах, связанных с графами, требуется организовать систематический обход всех вершин графа. Существуют два наиболее часто используемых метода обхода графов: поиск в глубину и поиск в ширину, о которых речь пойдет в этом разделе. Оба этих метода можно эффективно использовать для поиска вершин, смежных с данной вершиной.

Поиск в глубину

Напомним, что в разделе 6.5 мы построили алгоритм *dfs* для обхода вершин ориентированного графа. Этот же алгоритм можно использовать для обхода вершин и неориентированных графов, поскольку неориентированное ребро (v, w) можно представить в виде пары ориентированных дуг $v \to w$ и $w \to v$.

Фактически построить глубинный остовный лес для неориентированного графа (т.е. совершить обход его вершин) даже проще, чем для ориентированного. Во-первых, заметим, что каждое дерево остовного леса соответствует одной связной компоненте исходного графа, поэтому, если граф связный, его глубинный остовный лес будет состоять только из одного дерева. Во-вторых, при построении остовного леса для орграфа мы различали четыре типа дуг: дуги дерева, передние, обратные и поперечные дуги, а для неориентированного графа в этой ситуации выделяют только два типа ребер: ребра дерева и обратные ребра. Так как для неориентированного графа не существует направления ребер, то, естественно, прямые и обратные ребра здесь не различаются и объединены общим названием *обратные ребра*. Аналогично, так как в остовном дереве для неориентированного графа вершины не делятся на потомков и предков, то нет и перекрестных ребер. В самом деле, пусть есть ребро (v, w) и предположим, что при обходе графа вершина v достигнута раньше, чем вершина w. Тогда процедура $dfs(v)$не может закончиться раньше, чем будет рассмотрена вершина w. Поэтому в остовном дереве вершину w можно считать потомком вершины v. Но подобным образом, если сначала вызывается $dfs(w)$,вершина v становится потомком вершины w.

Итак, при обходе вершин неориентированного графа методом поиска в глубину все ребра делятся на следующие группы.

1. *Ребра дерева* — это такие ребра (v, w), что при обходе графа процедура $dfs(v)$ вызывается непосредственно перед процедурой $dfs(w)$ или, наоборот, сначала вызывается процедура $dfs(w)$,а затем сразу процедура $dfs(v)$.
2. *Обратные ребра* — такие ребра (v, w), что ни процедура $dfs(w)$не следует непосредственно за процедурой $dfs(v)$, ни процедура $dfs(v)$не следует непосредственно за процедурой $dfs(w)$(т.е. между вызовами этих процедур следуют вызовы нескольких других процедур $dfs(x)$).[1]

Пример 7.6. Рассмотрим связный граф G на рис. 7.8,а. Остовное дерево для этого графа, полученное методом поиска в глубину, показано на рис. 7.8,б. Поиск был начат с вершины а, и мы придерживались соглашения изображать ребра дерева сплошными линиями, а обратные ребра — пунктирными. Изображение остовного дерева начато от корня, и сыновья каждой вершины рисовались слева направо в том порядке, в каком они впервые посещались в процедуре *dfs*.

Опишем несколько шагов поиска в глубину для данного графа. Процедура $dfs(a)$ добавляет ребро (а, *b*) в остовное дерево *T*, поскольку вершина *b* ранее не посещалась, и вызывает процедуру *dfs(b)*. Процедура *dfs(b)* добавляет ребро (b, d) в остовное дере-

[1] Здесь приведены конструктивные определения ребер дерева и обратных ребер. Можно дать следующие, менее конструктивные, но более общие, определения: если при обходе графа достигнута вершина v, имеющая инцидентное ребро (v, w), то в случае, когда вершина w еще не посещалась во время этого обхода, данное ребро является ребром дерева, если же вершина w уже посещалась ранее, то ребро (v, w) — обратное ребро. — *Прим. ред.*

во Г и в свою очередь вызывает процедуру *dfs(d)*. Далее процедура *dfs(d)* добавляет в остовное дерево ребро *(d, e)* и вызывает *dfs(e)*. С вершиной *e* смежны вершины *a, Ь* и *d,* но они помечены как посещенные вершины. Поэтому процедура *dfs(e)* заканчивается без добавления ребер в дерево *Т.* Процедура *dfs(d)* также находит среди вершин, смежных с вершиной *d,* только вершины, помеченные как "посещенные". Процедура *dfs(d)* завершается без добавления новых ребер в остовное дерево. Процедура *dfs(b)* проверяет оставшиеся смежные вершины *a к e* (до этого была проверена только вершина *d).* Эти вершины посещались ранее, поэтому процедура *dfs(b)* заканчивается без добавления новых ребер в дерево *Т.* Процедура *dfs(a)*, продолжая работу, находит новые вершины, которые ранее не посещались. Это вершины *c, f* и *g.* □

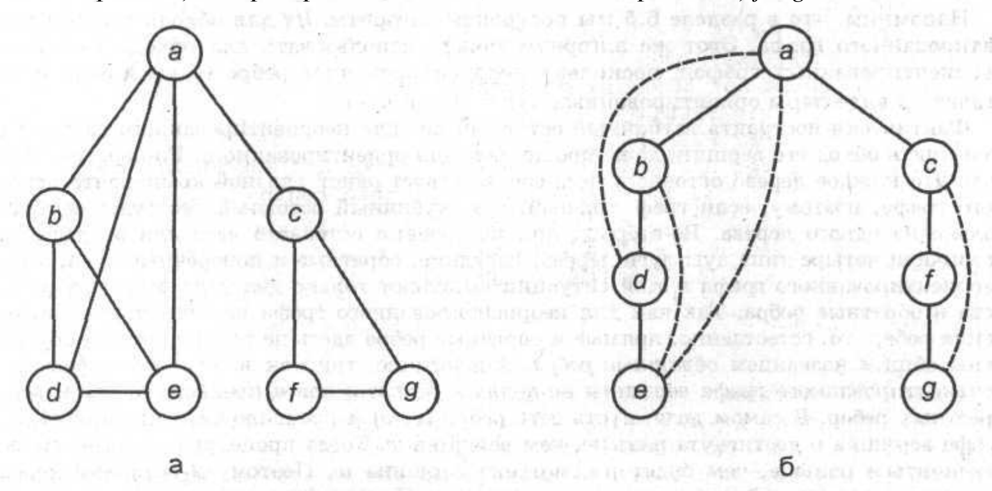


Рис. 7.8. Граф и остовное дерево, полученное при обходе его вершин методом поиска в глубину

Поиск в ширину

Другой метод систематического обхода вершин графа называется *поиском в ширину*. Он получил свое название из-за того, что при достижении во время обхода любой вершины *v* далее рассматриваются все вершины, смежные с вершиной *v,* т.е. осуществляется просмотр вершин "в ширину". Этот метод также можно применить и к ориентированным графам.

Так же, как и при применении поиска вглубь, посредством метода поиска в ширину при обходе графа создается остовный лес. Если после достижения вершины *x* при рассмотрении ребра *(x, y)* вершина *y* не посещалась ранее, то это ребро считается ребром дерева. Если же вершина *y* уже посещалась ранее, то ребро *(x. y)* будет поперечным ребром, так как оно соединяет вершины, не связанные наследованием друг друга.

В эскизе процедуры *bfs*[1] (листинг 7.4), реализующей алгоритм поиска в ширину, ребра дерева помещаются в первоначально пустой массив *T*, формирующий остовный лес. Посещенные вершины графа заносятся в очередь *Q*. Массив *mark* (метка) отслеживает состояние вершин: если вершина *v* пройдена, то элемент *mark[v]* принимает значение *visited* (посещалась), первоначально все элементы этого массива имеют значение *unvisited* (не посещалась). Отметим, что в этом алгоритме во избежание повторного помещения вершины в очередь пройденная вершина помечается как *visited* до помещения ее в очередь. Процедура *bfs* работает на одной связной компоненте, ес-

---

[1] Название процедуры *bfs* является сокращением от breadth-first search, что обозначает "поиск в ширину". — *Прим. ред.*

ли граф не **односвязный,** то эта **процедура** должна вызываться для вершин каждой связной компоненты.

Листинг 7.4. Алгоритм поиска в ширину

```
procedure bfs ( v );
    { bfs обходит все вершины, достижимые из вершины v }
    var
        Q: QUEUE { очередь для вершин }
        к, y: вершина;
    begin
        mark[v]:= visited;
        ENQUEUE(v, Q);
        while not EMPTY(Q) do begin
            x:= FRONT(Q);
            DEQUEUE(Q);
            for для каждой вершины y, смежной с вершиной x do
                if mark[y] = unvisited then begin
                    mark[y]:= visited;
                    ENQUEUE(y, Q);
                    INSERT((x, y), T)
                end
        end
    end; { bfs }
```

Пример 7.7. Остовное дерево для графа из рис. **7.8,а,** построенное методом поиска в ширину, показано на рис. 7.9. Здесь обход графа начат с вершины *а*. Как и ранее, ребра дерева показаны сплошными линиями, а другие ребра — пунктирными. Отметим также, что дерево нарисовано от корня, а все сыновья любой вершины располагаются слева направо в порядке посещения их. Р

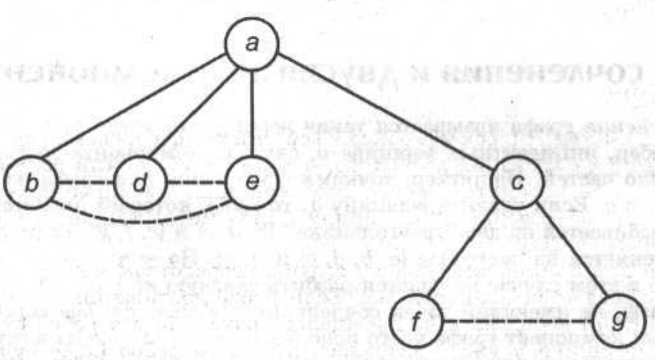


Рис. 7.9. Остовное дерево для графа из рис. 7.8,а, полученное методом поиска в ширину

Время выполнения алгоритма поиска в ширину такое **же,** как и для алгоритма поиска в глубину. Каждая пройденная вершина помещается в очередь только один раз, поэтому количество выполнений цикла while совпадает с количеством просмотренных вершин. Каждое ребро *(x, y)* просматривается дважды, один раз для вершины *x* и один раз для вершины *y*. Поэтому, если граф имеет *n* вершин и *е* ребер, а также если для представления ребер используются списки смежности, общее время обхода такого графа составит **$O(\max(n, e))$**. Поскольку обычно $e > n$, то получаем время выполнения алгоритма поиска в ширину порядка $O(e)$, т.е. такое же, как и для алгоритма поиска в глубину.

Методы поисков в глубину и в ширину часто используются как основа при разработке различных эффективных алгоритмов работы с графами. Например, оба этих метода можно применить для нахождения связных компонент графа, поскольку связные компоненты представимы отдельными деревьями в остовном лесу графа.

Метод поиска в ширину можно использовать для обнаружения циклов в произвольном графе, причем за время $O(n)$ (n — количество вершин графа), которое не зависит от количества ребер. Как показано в разделе 7.1, граф с п вершинами и с п или большим количеством ребер обязательно имеет цикл. Однако если в графе п - 1 или меньше ребер, то цикл может быть только в том случае, когда граф состоит из двух или более связных компонент. Один из способов нахождения циклов состоит в построении остовного леса методом поиска в ширину. Тогда каждое поперечное ребро *(v, w)* замыкает простой цикл, состоящий из ребер дерева, ведущих к вершинам *v* и *w* от общего предка этих вершин, как показано на рис. 7.10.

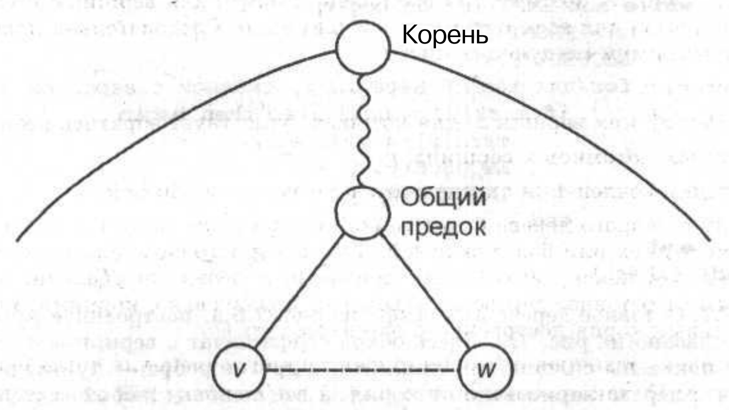


Рис. 7.10. Цикл, найденный методом поиска в ширину

7.4. Точки сочленения и двусвязные компоненты

Точкой сочленения графа называется такая вершина *v*, когда при удалении этой вершины и всех ребер, инцидентных вершине *v*, связная компонента графа разбивается на две или несколько частей. Например, точками сочленения для графа из рис. 7.8,а являются вершины а и с. Если удалить вершину *a*, то граф, который является одной связной компонентой, разбивается на два "треугольника" *{b, d, e}* и *{c, f, g}*. Если удалить вершину *c*, то граф расчленяется на подграфы *{a, b, d, e}* и *{/, g}*. Но если удалить какую-либо другую вершину, то в этом случае не удастся разбить связную компоненту на несколько частей. Связный граф, не имеющий точек сочленения, называется *двусвязным*. Для нахождения двусвязных компонент графа часто используется метод поиска в глубину.

Метод нахождения точек сочленения часто применяется для решения важной проблемы, касающейся *k-связности* графов. Граф называется *k-связным*, если удаление любых *k* - 1 вершин не приведет к расчленению графа.[1] В частности, граф имеет связность 2 или выше тогда и только тогда, когда он не имеет точек сочленения, т.е. только тогда, когда он является двусвязным. Чем выше связность графа, тем больше можно удалить вершин из этого графа, не нарушая его целостности, т.е. не разбивая его на отдельные компоненты.

---

[1] Существует другое, более конструктивное определение fe-связности. Граф называется *k*-связным, если между любой парой вершин *v* и *w* существует не менее *k* разных путей, таких, что, за исключением вершин *v* и *w*, ни одна из вершин, входящих в один путь, не входит ни в какой другой из этих путей. — *Прим. ред.*

Опишем простой алгоритм нахождения всех точек сочленения связного графа, основанный на методе поиска в глубину. При отсутствии этих точек граф, естественно, будет двусвязным, т.е. этот алгоритм можно рассматривать так же, как тест на двусвязность неориентированного графа.

1. Выполняется обход графа методом поиска в глубину, при этом для всех вершин v вычисляются числа *dfnumber*[v], введенные в разделе 6.5. В сущности, эти числа фиксируют последовательность обхода вершин в прямом порядке вершин глубинного остовного дерева.
2. Для каждой вершины v вычисляется число *low*[v], равное минимуму чисел *dfnumber* потомков вершины v, включая и саму вершину v, и предков w вершины v, для которых существует обратное ребро (x, w), где x — потомок вершины v. Числа *low*[v] для всех вершин v вычисляются при обходе остовного дерева в обратном порядке, поэтому при вычислении *low*[v] для вершины v уже подсчитаны числа *low*[x] для всех потомков x вершины v. Следовательно, *low*[v] вычисляется как минимум следующих чисел:
   а) *dfnumber*[v];
   б) *dfnumber*[z] всех вершин z, для которых существует обратное ребро (v, z);
   в) *low*[x] всех потомков x вершины v.
3. Теперь точки сочленения определяются следующим образом:
   а) корень остовного дерева будет точкой сочленения тогда и только тогда, когда он имеет двух или более сыновей. Так как в остовном дереве, которое получено методом поиска вглубь, нет поперечных ребер, то удаление такого корня расчленит остовное дерево на отдельные поддеревья с корнями, являющимися сыновьями корня построенного остовного дерева;
   б) вершина v, отличная от корня, будет точкой сочленения тогда и только тогда, когда имеет такого сына w, что *low*[w] $\geq$ *dfnumber*[v]. В этом случае удаление вершины v (и, конечно, всех инцидентных ей ребер) отделит вершину w и всех ее потомков от остальной части графа. Если же *low*[w] < *dfnumber*[v], то существует путь по ребрам дерева к потомкам вершины w и обратное ребро от какого-нибудь из этих потомков к истинному предку вершины v (именно значению *dfnumber* для этого предка равно *low*[w]). Поэтому в данном случае удаление вершины v не отделит от графа поддерево с корнем w.

Пример 7.8. Числа *dfnumber* и *low*, подсчитанные для вершин графа из рис. 7.8,а, показаны на рис. 7.11. В этом примере вычисление чисел *low* начато в обратном порядке с вершины e. Из этой вершины выходят обратные ребра (e, a) и (e, b), поэтому *low*[e] = min(*dfnumber*[e], *dfnumber*[a], *dfnumber*[b]) = 1. Затем рассматривается вершина d, для нее *low*[d] находится как минимум чисел *dfnumber*[d], *low*[e] и *dfnumber*[a]. Здесь число *low*[e] участвует потому, что вершина e является сыном вершины d, а число *dfnumber*[a] — из-за того, что существует обратное ребро (d, a).

Для определения точек сочленения после вычисления всех чисел *low* просматривается каждая вершина остовного дерева. Корень a является точкой сочленения, так как имеет двух сыновей. Вершина c также является точкой сочленения — для ее сына, вершины f, выполняется неравенство *low*[f] $\geq$ *dfnumber*[c]. Другие вершины не являются точками сочленения. □

Время выполнения описанного алгоритма на графе с n вершинами и e ребрами имеет порядок $O(e)$. Читатель легко проверит, что время, затрачиваемое на выполнение каждого этапа алгоритма, зависит или от количества посещаемых вершин, или от количества ребер, инцидентных этим вершинам, т.е. в любом случае время выполнения, с точностью до константы пропорциональности, является функцией как количества вершин, так и количества ребер. Поэтому общее время выполнения алгоритма имеет порядок $O(n + e)$ или $O(e)$, что то же самое при $n \leq e$.

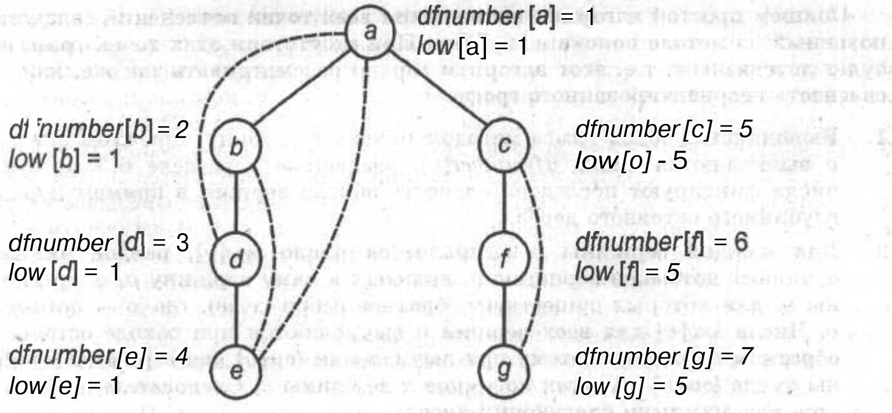


Рис. 7.11. Числа dfnumber *и* low, *подсчитанные для графа из рис. 7.8.а*

7.5. Паросочетания графов

В этом разделе мы опишем алгоритм решения "задачи о **паросочетании**" графов. Простым примером, приводящим к такой задаче, может служить ситуация распределения преподавателей по множеству учебных курсов. Надо назначить на чтение каждого курса преподавателя определенной квалификации так, чтобы ни на какой курс не было назначено более одного преподавателя. С другой стороны, желательно использовать максимально возможное количество преподавателей из их состава.

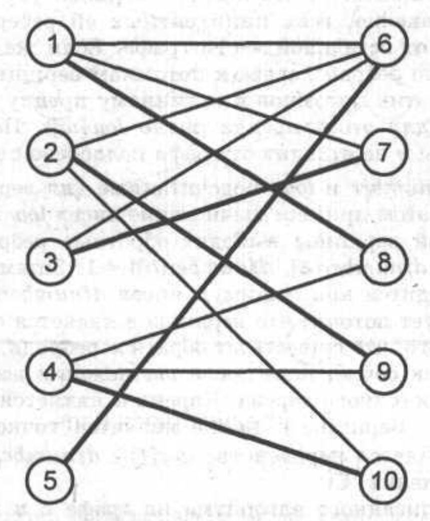


Рис. 7.12. Двудольный граф

Описанную ситуацию можно представить в виде графа, показанного на рис. 7.12, где все вершины разбиты на два множества V_1 и V_2 так, что вершины из множества V_1 соответствуют преподавателям, а вершины из множества V_2 — учебным курсам. Тот факт, что преподаватель v может вести курс w, отражается посредством ребра (v, w). Граф, у которого множество вершин распадается на два непересекающихся подмноже-

ства V_1 и V_2 таких, что каждое ребро графа имеет один конец из V_1, а другой — из V_2, называется *двудольным*. Таким образом, задача распределения преподавателей по учебным курсам сводится к задаче выбора определенных ребер в двудольном графе, имеющем множество вершин-преподавателей и множество вершин-учебных курсов.

Задачу паросочетания можно сформулировать следующим образом. Есть граф G = *(V, E)*. Подмножество его ребер, такое что никакие два ребра из этого подмножества не инциденты какой-либо одной вершине из *V*, называется *паросочетанием*. Задача определения максимального подмножества таких ребер называется *задачей нахождения максимального паросочетания*. Ребра, выделенные толстыми линиями на рис. 7.12, составляют одно возможное максимальное паросочетание для этого графа. *Полным паросочетанием* называется паросочетание, в котором участвуют (в качестве концов ребер) все вершины графа. Очевидно, что любое полное паросочетание является также максимальным паросочетанием.

Существуют прямые способы нахождения максимальных паросочетаний. Например, можно последовательно генерировать все возможные паросочетания, а затем выбрать то, которое содержит максимальное количество ребер. Но такой подход имеет существенный недостаток — время его выполнения является экспоненциальной функцией от числа вершин.

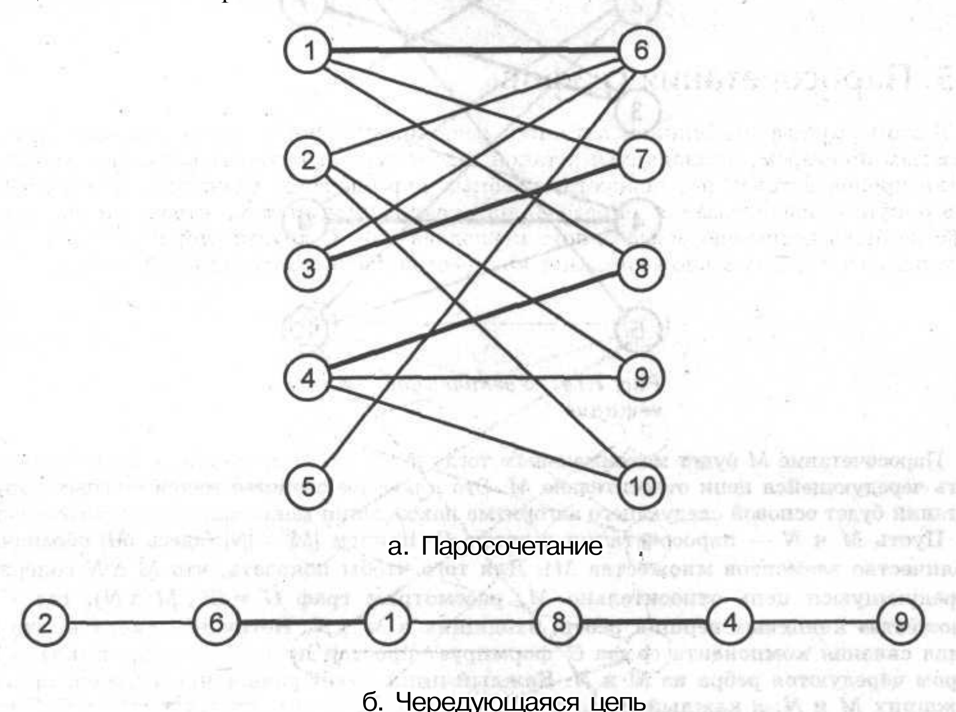


Рис. 7.13. Паросочетание и чередующаяся цепь

В настоящее время разработаны более эффективные алгоритмы нахождения максимальных паросочетаний. Эти алгоритмы используют в основном метод, известный как "чередующиеся цепи". Пусть *М* — паросочетание в графе G. Вершину *v* будем называть *парной,* если она является концом одного из ребер паросочетания *М*. Путь, соединяющий две непарные вершины, в котором чередуются ребра, входящие и не входящие в множество *М,* называется *чередующейся (аугментальной) цепью относительно М*. Очевидно, что чередующаяся цепь должна быть нечетной длины, начинаться и заканчиваться ребрами, не входящими в множество *М*. Также ясно, что,

имея чередующуюся цепь *Р,* мы можем увеличить паросочетание *М,* удалив из него те ребра, которые входят в цепь *Р,* и вместо удаленных ребер добавить ребра из цепи *Р,* которые первоначально не входили в паросочетание *М.* Это новое паросочетание можно определить как *М Л Р,* где Д обозначает симметрическую разность множеств *М и Р,* т.е. в новое множество паросочетаний войдут те ребра, которые входят или в множество *М,* или в множество *Р,* но не в оба сразу.

Пример 7.9. На рис. 7.13,а показаны граф и паросочетание *М,* состоящее из ребер (на рисунке они выделены толстыми линиями) (1, 6), (3, 7) и (4, 8). На рис. 7.13,б представлена чередующаяся цепь относительно *М,* состоящая из вершин 2, 6, 1, 8, 4, 9. На рис. 7.14 изображено паросочетание (1, 8), (2, 6), (3, 7), (4, 9), которое получено удалением из паросочетания *М* ребер, входящих в эту чередующуюся цепь, и добавлением новых ребер, также входящих в эту цепь. □

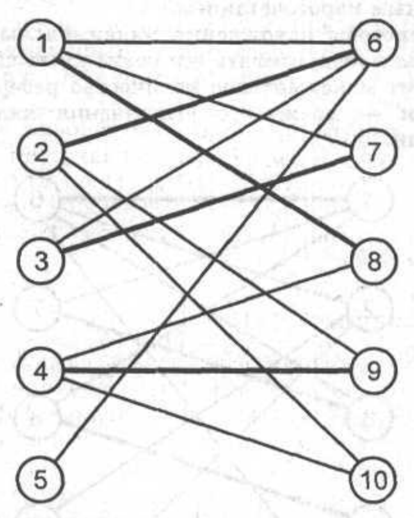


Рис. 7.14. Увеличенное паросочетание

Паросочетание *М* будет максимальным тогда и только тогда, когда не будет существовать чередующейся цепи относительно *М*. Это ключевое свойство максимальных паросочетаний будет основой следующего алгоритма нахождения максимального паросочетания.

Пусть *М и N* — паросочетания в графе *G,* причем *М*\ < *N*\ (здесь *М*\ обозначает количество элементов множества *М).* Для того чтобы показать, что *М Л N* содержит чередующуюся цепь относительно *М,* рассмотрим граф *G* = (*V*', *М* Д *N),* где *V* — множество концевых вершин ребер, входящих в *М* Д *N*. Нетрудно заметить, что каждая связная компонента графа G" формирует простой путь (возможно, цикл), в котором чередуются ребра из *М* и *N*. Каждый цикл имеет равное число ребер, принадлежащих *М* и *N,* а каждый путь, не являющийся циклом, представляет собой чередующуюся цепь относительно или *М,* или *N,* в зависимости от того, ребер какого паросочетания больше в этой цепи. Поскольку *М*\ < *N*\, то множество $M \Delta N$содержит больше ребер из паросочетания *N,* чем *М*, и, следовательно, существует хотя бы одна чередующаяся цепь относительно *М*.

Теперь можно описать алгоритм нахождения максимального паросочетания *М* для графа *G = (V, Е).*

1. Сначала положим *М = 0.*
2. Далее ищем чередующуюся цепь *Р* относительно *М,* и множество *М* заменяется на множество $M \Delta P$.

3. Шаг 2 повторяется до тех пор, пока существуют чередующиеся цепи относительно М. Если таких цепей больше нет, то M — максимальное паросочетание.

Нам осталось показать способ построения чередующихся цепей относительно паросочетания M. Рассмотрим более простой случай, когда граф G является двудольным графом с множеством вершин, разбитым на два подмножества V_1 *и* V_2. Мы будем строить *граф чередующейся цепи* по уровням i = 0, 1, 2, ..., **используя** процесс, подобный поиску в ширину. Граф чередующейся цепи уровня 0 содержит все непарные вершины из множества V_1. На уровне с нечетным номером i добавляются новые вершины, смежные с вершинами уровня i - 1 и соединенные ребром, не входящим в паросочетание (это ребро тоже добавляется в строящийся граф). На уровне с четным номером i также добавляются новые вершины, смежные с вершинами уровня i - 1, но которые соединены ребром, входящим в паросочетание (это ребро тоже добавляется в граф чередующейся цепи).

Процесс построения продолжается до тех пор, пока к графу чередующейся цепи можно присоединять новые вершины. Отметим, что непарные вершины присоединяются к этому графу только на нечетном уровне. В построенном графе путь от любой вершины нечетного уровня до любой вершины уровня 0 является чередующейся цепь относительно M.

Пример 7.10. На рис. 7.15 изображен граф чередующейся цепи, построенный для графа из рис. 7.13, а относительно паросочетания, показанного на рис. 7.14. На уровне 0 имеем одну непарную вершину 5. На уровне 1 добавлено ребро (5, 6), не входящее в паросочетание. На уровне 2 добавлено ребро (6, 2), входящее в паросочетание. На уровне 3 можно добавить или ребро (2, 9), или ребро (2, 10), не входящие в паросочетание. Поскольку вершины 9 и 10 пока в этом графе непарные, можно остановить процесс построения графа чередующейся цепи, добавив в него одну или другую вершину. Оба пути 9, 2, 6, 5 и 10, 2, 6, 5 являются чередующимися цепями относительно паросочетания из рис. 7.14. □

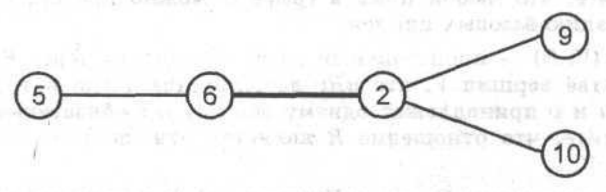


Рис. 7.15. Граф чередующейся цепи

Пусть граф G имеет n вершин и e ребер. Если используются списки смежности для представления ребер, то на построение графа чередующейся цепи потребуется времени порядка $O(e)$. Для нахождения максимального паросочетания надо построить не более $n/2$ чередующихся цепей, поскольку каждая такая цепь увеличивает текущее паросочетание не менее чем на одно ребро. Поэтому максимальное паросочетание для двудольного графа можно найти за время порядка $O(ne)$.

Упражнения

7.1. Опишите алгоритм вставки и удаления ребер неориентированного графа, представленного списками смежности. Помните, что каждое ребро (i, j) появляется в списке смежности вершины i и вершины j.

7.2. Измените представление неориентированного графа посредством списков смежности так, чтобы первое ребро в списке смежности любой вершины можно было бы удалить за фиксированное время. Напишите процедуру удаления первых ребер, инцидентным вершинам, используя новое представление списков смежности. *Подсказка:* как сделать так, чтобы две ячейки, соответствующие ребру (i, j), могли быстро находить друг друга?

7.3. Для графа, показанного на рис. 7.16, постройте

а) остовное дерево минимальной стоимости посредством алгоритма Прима;

б) остовное дерево минимальной стоимости с помощью алгоритма Крускала;

в) остовное дерево методом поиска в глубину, начиная с вершин a и d;

г) остовное дерево методом поиска в ширину, начиная с вершин a и d.

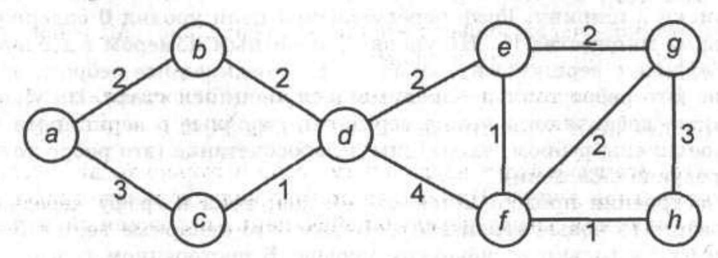


Рис. 7.16. Граф

7.4. Пусть T — глубинное остовное дерево и B — множество обратных ребер для связного неориентированного графа G = (V, E).

*а) Покажите, что добавление в остовное дерево T любого обратного ребра из множества B приводит к образованию в T одного цикла. Назовем такой цикл *базовым*.

**б) *Линейной комбинацией* циклов C_1, C_2, ..., С„ называется структура C_1 Д C_2 А ... А С„. Докажите, что линейная комбинация двух различных пересекающихся циклов является циклом.

**с) Покажите, что любой цикл в графе G можно представить как линейную комбинацию базовых циклов.

*7.5. Пусть G = (V, E) — произвольный граф. Обозначим через R такое отношение на множестве вершин V, что uRv выполняется тогда и только тогда, когда вершины u и v принадлежат одному общему (не обязательно простому) циклу. Докажите, что отношение Д является отношением эквивалентности на множестве V.

7.6. Реализуйте алгоритмы Прима и Крускала. Сравните время выполнения ваших программ на множестве "случайных" графов.

7.7. Напишите программу нахождения всех связных компонент графа.

7.8. Напишите программу с временем выполнения $O(n)$, чтобы определить, есть ли в графе с n вершинами цикл.

7.9. Напишите программу пересчета всех простых циклов в графе. Сколько может быть таких циклов? Какова временная сложность (время выполнения) такой программы?

7.10. Докажите, что при обходе вершин графа методом поиска в ширину каждое ребро может быть или ребром дерева, или обратным ребром.

7.11. Реализуйте алгоритм нахождения точек сочленения, описанный в разделе 7.4.

*7.12. Пусть G = (V, E) — *полный граф*, т.е. такой, что для любой пары различных вершин существует соединяющее их ребро. Пусть $G' = (V, E')$ — ориентированный граф, в котором множество дуг E' получено путем случайной ориентации ребер из множества E. Покажите, что орграф G' имеет путь, который проходит через все вершины графа точно один раз.

*7.13. Покажите, что полный граф с л вершина имеет n^{n-2} остовных деревьев.

7.14. Найдите все максимальные паросочетания для графа, изображенного на рис. 7.12.

, 7.15. Напишите программу нахождения максимального паросочетания для двудольного графа.

7.16. Пусть *M* — паросочетание и *m* — число ребер в максимальном паросочетании. Докажите, что

а) существуют чередующееся цепи относительно *M*, чья длина равна $2(|M|/(m - |M|)) + 1$ или меньше;

б) если *P* — кратчайшая чередующаяся цепь относительно *M*, *а P′* — чередующаяся цепь относительно *M* Д *P*, тогда |*P′*| > *P*\ + *P* Л *P′*|.

***7.17.** Докажите, что граф будет двудольным тогда и только тогда, когда он не имеет циклов нечетной длины. Приведите пример недвудольного графа, для которого способ построения графа чередующейся цепи, изложенный в разделе 7.5, не работает.

7.18. Пусть *M и N* — паросочетания в двудольном графе, причем $|M| < |N|$. Докажите, что $M \Delta N$ имеет по крайней мере *N*\ - |*M*| вершин, не входящих в чередующиеся цепи относительно ***M***.

Библиографические примечания

Изучение методов построения минимальных **остовных** деревьев начато Борувкой (Boruvka) еще *в* 1926 году [14]. Два описанных в этой главе алгоритма построения остовных деревьев взяты из [67] и [88]. В работе [57] показано, как можно использовать частично упорядоченные деревья для реализации алгоритма Прима. В [17] и [123] представлены алгоритмы построения остовных деревьев с временной сложностью $O(e \log \log n)$. В [109] предлагается исчерпывающий обзор и историческая справка по алгоритмам построения остовных деревьев.

В [52] и [107] рассмотрены различные варианты применения метода поиска в глубину для алгоритмов, работающих с графами. Алгоритм нахождения двусвязных компонент взят из этих работ.

Паросочетания графов изучались в [46], а чередующиеся цепи — в [10] и [27]. В работе [51] приведен алгоритм нахождения максимального паросочетания для двудольных графов с временной сложностью $O(n^{2.5})$, а в [75] — с временем выполнения $O(\sqrt{|V|} \cdot |E|)$ для произвольных графов. Статья [82] содержит хорошее обсуждение проблем паросочетании.

ГЛАВА 8

Сортировка

Сортировкой, или упорядочиванием списка объектов называется расположение этих объектов по возрастанию или убыванию согласно определенному линейному отношению порядка, такому как отношение "$<$" для чисел. Это очень большая тема, поэтому она разбита на две части: внутреннюю и внешнюю сортировку. При внутренней сортировке все сортируемые данные помещаются в оперативную память компьютера, где можно получить доступ к данным в любом порядке (т.е. используется модель памяти с произвольным доступом). Внешняя сортировка применяется тогда, когда объем упорядочиваемых данных слишком большой, чтобы все данные можно было поместить в оперативную память. Здесь узким местом является механизм перемещения больших блоков данных от устройств внешнего хранения данных к оперативной памяти компьютера и обратно. Тот факт, что физически непрерывные данные надо для удобного перемещения организовывать в блочную структуру, заставляет нас применять разнообразные методы внешней сортировки. Эти методы будут рассмотрены в главе 11.

8.1. Модель внутренней сортировки

В этой главе мы представим основные принципиальные алгоритмы внутренней сортировки. Простейшие из этих алгоритмов затрачивают время порядка $O(n^2)$ для упорядочивания n объектов и потому применимы только к небольшим множествам объектов. Один из наиболее популярных алгоритмов сортировки, так называемая быстрая сортировка, выполняется в среднем за время $O(n \log n)$. Быстрая сортировка хорошо работает в большинстве приложений, хотя в самом худшем случае она также имеет время выполнения $O(n^2)$. Существуют другие методы сортировки, такие как пирамидальная сортировка или сортировка слиянием, которые в самом худшем случае дают время порядка $O(n \log n)$, но в среднем (в статистическом смысле) работают не лучше, чем быстрая сортировка. Отметим, что метод сортировки слиянием хорошо подходит для построения алгоритмов внешней сортировки. Мы также рассмотрим алгоритмы другого типа, имеющие общее название "карманной" сортировки. Эти алгоритмы работают только с данными определенного типа, например с целыми числами из ограниченного интервала, но когда их можно применить, то они работают очень быстро, затрачивая время порядка $O(n)$ в самом худшем случае.

Всюду в этой главе мы будем предполагать, что сортируемые объекты являются записями, содержащими одно или несколько полей. Одно из полей, называемое *ключом,* имеет такой тип данных, что на нем определено отношение линейного порядка "$\leq$". Целые и действительные числа, символьные массивы — вот общие примеры таких типов данных, но, конечно, мы можем использовать ключи других типов данных, лишь бы на них можно было определить отношение "меньше чем" или "меньше чем или равно".

Задача сортировки состоит в упорядочивании последовательности записей таким образом, чтобы значения ключевого поля составляли неубывающую последовательность. Другими словами, записи r_1, r_2, ...» r_n со значениями ключей $k_1, k_2, ..., k_n$ надо расположить в порядке $r_{i_1}, r_{i_2}, ..., r_{i}$, таком, что $k_{i_1} \leq k_{i_2} \leq ... < k_{i}$. Мы не требуем, чтобы все записи были различными, и если есть записи с одинаковыми значениями ключей, то в упорядоченной последовательности они располагаются рядом друг с другом в любом порядке.

Мы будем использовать различные критерии оценки времени выполнения алгоритмов внутренней сортировки. Первой и наиболее общей мерой времени выполнения является количество шагов алгоритма, необходимых для упорядочивания л записей. Другой общей мерой служит количество сравнений между значениями ключей, выполняемых при сортировке списка из *n* записей. Эта мера особенно информативна, когда ключи являются строками символов, и поэтому самым "трудоемким" оператором будет оператор сравнения ключей. Если размер записей большой, то следует также учитывать время, необходимое на перемещение записей. При создании конкретных приложений обычно ясно, по каким критериям нужно оценивать применяемый алгоритм сортировки.

8.2. Простые схемы сортировки

По-видимому, самым простым методом сортировки является так называемый *метод "пузырька"*. Чтобы описать основную идею этого метода, представим, что записи, подлежащие сортировке, хранятся в массиве, расположенном вертикально. Записи с малыми значениями ключевого поля более "легкие" и "всплывают" вверх наподобие пузырька. При первом проходе вдоль массива, начиная проход снизу, берется первая запись массива и ее ключ поочередно сравнивается с ключами последующих записей. Если встречается запись с более "тяжелым" ключом, то эти записи меняются местами. При встрече с записью с более "легким" ключом эта запись становится "эталоном" для сравнения, и все последующие записи сравниваются с этим новым, более "легким" ключом. В результате запись с наименьшим значением ключа окажется в самом верху массива. Во время второго прохода вдоль массива находится запись со вторым по величине ключом, которая помещается под записью, найденной при первом проходе массива, т.е. на вторую сверху позицию, и т.д. Отметим, что во время второго и последующих проходов вдоль массива нет необходимости просматривать записи, найденные за предыдущие проходы, так как они имеют ключи, меньшие, чем у оставшихся записей. Другими словами, во время *i*-го прохода не проверяются записи, стоящие на позициях выше *i*. В листинге 8.1 приведен описываемый алгоритм, в котором через *A* обозначен массив из п записей (тип данных recordtype). Здесь и далее в этой главе предполагаем, что одно из полей записей называется *key* (ключ) и содержит значения ключей.

Листинг 8.1. Алгоритм „пузырька"

```
(1)     for i:= 1 to n - 1 do
(2)         for j: = 1 downto i + 1 do
(3)             if A[j].key< A[j - 1].key then
(4)                 swap(A[j], A[j - 1])
```

Процедура *swap* (перестановка) используется во многих алгоритмах сортировки для перестановки записей местами, ее код показан в следующем листинге.

Листинг 8.2. Процедура *swap*

```
procedure swap ( var x, y: recordtype )
    { swap меняет местами записи x и y }
    var
        temp: recordtype;
    begin
        temp:= x;
        x:= y;
        y:= temp;
    end; { swap }
```

Пример 8.1. В табл. 8.1 приведен список названий и годы знаменитых извержений вулканов.

Таблица 8.1. Знаменитые извержения вулканов

Название	Год
Пили	1902
Этна	1669
Кракатау	1883
Агунг	1963
Св. Елена	1980
Везувий	79

Для этого примера мы применим следующее объявление типов данных:

```
type
    keytype = array[1..10] of char;
    recordtype = record
        key: keytype; { название вулкана }
        year: integer { год извержения }
    end;
```

Применим алгоритм "пузырька" для упорядочивания списка вулканов в алфавитном порядке их названий, считая, что отношением линейного порядка в данном случае является отношение лексикографического порядка над полем ключей. В табл. 8.2 показаны пять проходов алгоритма (в данном случае $n = 6$). Линии указывают позицию, выше которой записи уже упорядочены. После пятого прохода все записи, кроме последней, стоят на нужных местах, но последняя запись не случайно оказалась последней — она также уже стоит на нужном месте. Поэтому сортировка заканчивается.

Таблица 8.2. Сортировка методом „пузырька"

Начальное положение	1-й проход	2-й проход	3-й проход	4-й проход	5-й проход
Пили	Агунг	Агунг	Агунг	Агунг	Агунг
Этна	Пили	Везувий	Везувий	Везувий	Везувий
Кракатау	Этна	Пили	Кракатау	Кракатау	Кракатау
Агунг	Кракатау	Этна	Пили	Пили	Пили
Св. Елена	Везувий	Кракатау	Этна	Св. Елена	Св. Елена
Везувий	Св. Елена	Св. Елена	Св. Елена	Этна	Этна

На первом проходе Везувий и Св. Елена меняются местами, но далее Везувий не может поменяться местами с Агунгом. Агунг, поочередно меняясь местами с Кракатау, Этной и Пили, поднимается на самый верх. На втором этапе Везувий поднимается вверх и занимает вторую позицию. На третьем этапе это же проделывает Кракатау, а на четвертом — Этна и Св. Елена меняются местами, Пили стоит на нужном месте. Все названия вулканов расположились в алфавитном порядке, сортировка закончилась. Пятый этап также выполнен в соответствии с алгоритмом листинга 8.1, но он уже не меняет порядок записей. П

Сортировка вставками

Второй метод, который мы **рассмотрим,** называется сортировкой вставками, так как на i-м этапе мы "вставляем" *i*-й элемент *A[i]* в нужную позицию среди элементов A[1], *A[2]*, ..., **A[*i*** - 1], которые уже упорядочены. После этой вставки первые i элементов будут упорядочены. Сказанное можно записать в виде следующей псевдопрограммы:

```
for i:= 2 to л do
        переместить A[i] на позицию j ≤ i такую, что
            A[i] < A[k] для j ≤ k < i и
            либо A[i] ≥ A[j - 1], либо j = 1
```

Чтобы сделать процесс перемещения элемента ***A*[*i*]** более простым, полезно ввести элемент A[0], чье значение ключа будет меньше значения ключа любого элемента A[1], ..., ***A*[*n*]**. Мы можем постулировать существование константы - ∞ типа keytype, которая будет меньше значения ключа любой записи, встречающейся на практике. Если такую константу нельзя применить, то при вставке *A[i]* в позицию /' - 1 надо проверить, не будет ли *j* = 1, если нет, тогда сравнивать элемент ***A*[*i*]** (который сейчас находится в позиции *j*) с элементом ***A*[*j*** - 1]. Описанный алгоритм показан в листинге 8.3.

Листинг 8.3. Сортировка вставками

```
(1)     A[0] .key:= - ∞;
(2)     for i:= 2 to n do begin
(3)         j:= i;
(4)         while A[j] < A[j - 1] do begin
(5)             swap(A[j], A[j - 1]);
(6)             j:= j - 1
            end
        end
```

Пример 8.2. В табл. 8.3 показан начальный список из табл. 8.1 и последовательные этапы алгоритма вставкой для *i* = *2*, 3, ..., 6. После каждого этапа алгоритма элементы, расположенные выше линии, уже упорядочены, хотя между ними на последующих этапах могут быть вставлены элементы, которые сейчас находятся ниже линии. D

Таблица 8.3. Этапы сортировки вставками

— ∞	— ∞	— ∞	— ∞	— ∞	— ∞
Пили	Пили	Кракатау	Агунг	Агунг	Агунг
Этна	Этна	Пили	Кракатау	Кракатау	Везувий
Кракатау	Кракатау	Этна	Пили	Пили	Кракатау
Агунг	Агунг	Агунг	Этна	Св. **Елена**	Пили
Св. **Елена**	Св. Елена	Св. Елена	Св. Елена	Этна	Св. Елена
Везувий	Везувий	Везувий	Везувий	Везувий	Этна
Начальное положение	i = 2	*i* = 3	i = 4	*i* = 5	*i* = 6

Сортировка посредством выбора

Идея сортировки посредством выбора также элементарна, как и те два метода сортировки, которые мы уже рассмотрели. На i-м этапе сортировки выбирается запись с наименьшим ключом среди записей **A[i], ..., A[n]** и меняется местами с записью **A[i]**. В результате после **i-го** этапа все записи **A[1], ..., A[i]** будут упорядочены. Сортировку посредством выбора можно описать следующим образом:

```
for i:= 1 to n - 1 do
     выбрать среди A[i], ..., A[л] элемент с наименьшим ключом и
          поменять его местами с A[i];
```

Более полный код, реализующий этот метод сортировки, приведен в листинге 8.4.

Листинг 8.4. Сортировка посредством выбора

```
        var
            lowkey: keytype; { текущий наименьший ключ, найденный
                 при проходе по элементам A[i], ..., A[n] }
            lowindex: integer; { позиция элемента с ключом lowkey }
        begin
(1)         for i:= 1 to л - 1 do begin
(2)             lowindex:= i;
(3)             lowkey:= A[i].key;
(4)             for j:= i + 1 to л do
                    { сравнение ключей с текущим ключом lowkey }
(5)                 if A[j].key < lowkey then begin
(6)                     lowkey:= A[j].key;
(7)                     lowindex:= j
                    end;
(8)             swap(A[i], A[lowindex])
            end
        end;
```

Пример 8.3. В табл. 8.4 показаны этапы сортировки посредством выбора для списка из табл. 8.1. Например, на 1-м этапе значение *lowindex* равно 4, т.е. позиции **Агунга,** который меняется с Пили, элементом **A[1]**.

Линии в табл. 8.4 показывают, что элементы, расположенные выше ее, имеют наименьшие значения ключей и уже упорядочены. После (п - 1)-го этапа элемент **A[n]** также стоит на "правильном" месте, так как выше его все записи имеют меньшие значения ключей. D

Таблица 8.4. Сортировка посредством выбора

Начальное положение	1-й этап	2-й этап	3-й этап	4-й этап	5-й этап
Пили	Агунг	Агунг	Агунг	Агунг	Агунг
Этна	Этна	Везувий	Везувий	Везувий	Везувий
Кракатау	Кракатау	Кракатау	Кракатау	Кракатау	Кракатау
Агунг	Пили	Пили	Пили	Пили	Пили
Св. Елена	Св. Елена	Св. Елена	Св. Елена	Св. Елена	Св. Елена
Везувий	Везувий	Этна	Этна	Этна	Этна

Временная сложность методов сортировки

Методы "пузырька", вставками и посредством выбора имеют временную сложность $O(n^2)$ и $\Omega(n^2)$ на последовательностях из п элементов. Рассмотрим метод "пузырька" (листинг 8.1). Независимо от того, что подразумевается под типом **recordtype, выполнение** процедуры *swap* требует фиксированного времени. Поэтому строки (3), (4) затрачивают c_1 единиц времени; c_1 — некоторая константа. Следовательно, для фиксированного значения i цикл строк (2) - (4) требует не больше $c_2(n - i)$ шагов; c_2 — константа. Последняя константа несколько больше константы c_1, если учитывать операции с индексом *j* в строке (2). Поэтому вся программа требует

$$c_3n + \sum_{i=1}^{n-1} c_2(n - i) = \frac{1}{2}c_2n^2 + (c_3 - \frac{1}{2}c_2)n$$

шагов, где слагаемое c_3n учитывает операции с индексом *i в* строке (1). Последнее выражение не превосходит $(c_2/2 + c_3)n^2$ для $n > 1$, поэтому алгоритм "пузырька" имеет временную сложность $O(n^2)$. Нижняя временная граница для алгоритма равна $\Omega(n^2)$, поскольку если даже не выполнять процедуру *swap* (например, если список уже отсортирован), то все равно $n(n - 1)/2$ раз выполняется проверка в строке (3).

Далее рассмотрим сортировку вставками (листинг 8.3). Цикл while в строках (4) - (6) выполняется не более *O(i)* раз, поскольку начальное значение у равно *i, а* затем *j* уменьшается на 1 при каждом выполнении этого цикла. Следовательно, цикл **for** строк (2) - (6) потребует не более $c\sum_{i=2}^{n} i$ шагов для некоторой константы *с*. Эта сумма имеет порядок $O(n^2)$.

Читатель может проверить, что если список записей первоначально был отсортирован в обратном порядке, то цикл while в строках (4) - (6) выполняется ровно i - 1 раз, поэтому строка (4) выполняется $\sum_{i=2}^{n}(i-1) = n(n-1)/2$ раз. Следовательно, сортировка вставками в самом худшем случае требует времени не менее $\Omega(n^2)$. Можно показать, что нижняя граница в среднем будет такой же.

Наконец, рассмотрим сортировку посредством выбора, показанную в листинге 8.4. Легко проверить, что внутренний цикл в строках (4) - (7) требует времени порядка $O(n - i)$, поскольку *j* здесь изменяется от i + 1 до и. Поэтому общее время выполнения алгоритма составляет $c\sum_{i=1}^{n-1}(n - i)$ для некоторой константы *с*. Эта сумма, равная $cn(n - 1)/2$, имеет порядок роста $O(n^2)$. С другой стороны, нетрудно показать, что строка (4) выполняется не менее $\sum_{i=1}^{n-1}\sum_{j=i+1}^{n}(1) = n(n-1)/2$ раз независимо от начального списка сортируемых элементов. Поэтому сортировка посредством выбора требует времени не менее $\Omega(n^2)$ в худшем случае и в среднем.

Подсчет перестановок

Если размер записей большой, тогда процедура swap для перестановки записей, которая присутствует во всех трех вышеописанных алгоритмах, занимает больше времени, чем все другие операции, выполняемые в программе (например, такие как сравнение ключей или вычисление индексов массива). Поэтому, хотя мы уже показали, что время работы всех трех алгоритмов пропорционально n^2, необходимо более детально сравнить их с учетом использования процедуры *swap*.

Сначала рассмотрим алгоритм "пузырька". В этом алгоритме процедура *swap* выполняется (см. листинг 8.1, строка (4)) не менее $\sum_{i=1}^{n-1}\sum_{j=i+1}^{n}(1) = n(n-1)/2$ раз, т.е. почти $n^2/2$ раз. Но поскольку строка (4) выполняется после условного оператора в строке (3), то можно ожидать, что число перестановок значительно меньше, чем $n^2/2$.

В самом деле, в среднем перестановки выполняются только в половине из всех возможных случаев. Следовательно, ожидаемое число перестановок, если все воз-

можные исходные последовательности ключей **равновероятны,** составляет примерно $n^2/4$. Для доказательства этого утверждения рассмотрим два списка ключей, которые обратны друг другу: $L_1 = k_1, k_2, \ldots, k_n$ и $L_2 = k_n$, ftв_i, ..., fej. Переставляются ключи k_i и k_j из одного списка, если они расположены в "неправильном" порядке, т.е. нарушают отношение линейного порядка, заданного на множестве значений ключей. Такая перестановка для ключей k_i и k_j выполняется только один раз, так как они расположены неправильно или только в списке L_1, или только в списке L_2, но не в обоих сразу. Поэтому общее число перестановок в алгоритме "пузырька", примененном к спискам L_1 и L_2, равно числу пар элементов, т.е. числу сочетаний из n по 2: $C^* = n(n-1)/2$. Следовательно, среднее число перестановок для списков L_1 и L_2 равно $n(n-1)/4$ или примерно $n^2/4$. Поскольку для любого упорядочивания ключей существует обратное к нему, как в случае списков L_1 и L_2, то отсюда вытекает, что среднее число перестановок для любого списка также равно примерно $n^2/4$.

Число перестановок для сортировки вставками в среднем точно такое же, как и для алгоритма "пузырька". Для доказательства этого достаточно применить тот же самый аргумент: каждая пара элементов подвергается перестановке или в самом упорядочиваемом списке L и в обратном к нему, но никогда в обоих.

Если на выполнение процедуры *swap* необходимы большие вычислительные или временные ресурсы, то легко увидеть, что в этом случае сортировка посредством выбора более предпочтительна, нежели другие рассмотренные методы сортировки. В самом деле, в алгоритме этого метода (листинг 8.4) процедура *swap* выполняется вне внутреннего цикла, поэтому она выполняется точно п - 1 раз для массива длиной п. Другими словами, в этом алгоритме осуществляется $O(n)$ перестановок в отличие от двух других алгоритмов, где количество таких перестановок имеет порядок $O(n^2)$. Причина этого понятна: в методе сортировки посредством выбора элементы "перескакивают" через большой ряд других элементов, вместо того чтобы последовательно меняться с ними местами, как это делается в алгоритме "пузырька" и алгоритме сортировки вставками.

В общем случае, если необходимо упорядочить длинные записи (поэтому их перестановки занимают много времени), целесообразно работать не с массивом записей, а с массивом указателей на записи, используя для сортировки любой подходящий алгоритм. В этом случае переставляются не сами записи, а только указатели на них. После упорядочивания указателей сами **записи** можно расположить в нужном порядке за время порядка $O(n)$.

Ограниченность простых схем сортировки

Еще раз напомним, что каждый из рассмотренных в этом разделе алгоритмов выполняется за время порядка $O(n^2)$ как в среднем, так и в самом худшем случае. Поэтому для больших п эти алгоритмы заведомо проигрывают алгоритмам с временем выполнения $O(n \log n)$, которые будут описаны в следующем разделе. Значение n, начиная с которого быстрые алгоритмы сортировки становятся предпочтительнее простых методов сортировки, зависит от различных факторов, таких как качество объектного кода программы, генерируемого компилятором, компьютера, на котором выполняется программа сортировки, или от размера записей. Определить такое критическое значение п для конкретных задач и вычислительного окружения (компилятор, компьютер и т.п.) можно после экспериментов с **профайлером**[1] и на основании выдаваемых им результатов. Практический опыт учит, что при га, не превышающем 100, на время выполнения программы влияет множество факторов, которые с трудом поддаются регистрации и поэтому не учтены в анализе, проведенном в этом разделе. Для небольших значений п рекомендуем применять простой в реализации алгоритм сортировки Шелла (Shell), который имеет **временную** сложность $O(n^{1.5})$. Этот алгоритм, описанный в упражнении 8.3, является обобщением алгоритма "пузырька".

[1] **Профайлер** — это подпрограмма **протоколирования,** позволяющая оценить время выполнения отдельных функций и операторов программы. — *Прим. ред.*

8.3. Быстрая сортировка

Первый алгоритм с временем выполнения $O(n \log n)$[1], который мы рассмотрим далее, является, по-видимому, самым эффективным методом внутренней сортировки и поэтому имеет название "быстрая сортировка".[2] В этом алгоритме для сортировки элементов массива $A[1]$, ..., $A[n]$ из этих элементов выбирается некоторое значение ключа v в качестве *опорного элемента,* относительно которого переупорядочиваются элементы массива. Желательно выбрать опорный элемент близким к значению медианы распределения значений ключей так, чтобы опорный элемент разбивал множество значений ключей на две примерно равные части. Далее элементы массива переставляются так, чтобы для некоторого индекса j все переставленные элементы $A[1]$, ..., $A[j]$ имели значения ключей, меньшие чем v, а все элементы $A[j+1]$, ..., $A[n]$ — значения ключей, большие или равные v. Затем процедура быстрой сортировки рекурсивно применяется к множествам элементов $A[1]$, ..., $A[j]$ и $A[j+1]$, ..., $A[n]$ для упорядочивания этих множеств по отдельности. Поскольку все значения ключей в первом множестве меньше, чем значения ключей во втором множестве, то исходный массив будет отсортирован правильно.

Пример 8.4. На рис. 8.9 показаны последовательные шаги выполнения алгоритма быстрой сортировки, выполняемые над последовательностью целых чисел 3, 1, 4, 1, 5, 9, 2, 6, 5, 3. На каждом этапе значение v выбирается как наибольшее значение из двух самых левых различных элементов-чисел. Сортировка завершается, когда отдельные частичные подмножества, на которые разбивается исходный массив в процессе рекурсивных вызовов процедуры, будут содержать одинаковые ключи. На рис. 8.9 каждый этап показан в виде двух шагов: сначала для каждого частичного подмножества выбирается свое значение v, затем элементы в этих подмножествах переставляются в соответствии с выбранным значением v, тем самым опять разбиваются еще на два подмножества, к которым снова рекурсивно применяется процедура сортировки. □

Теперь начнем разрабатывать рекурсивную процедуру *quicksort*(i, j), которая будет работать с элементами массива A, определенным вне этой процедуры. Процедура *quicksort*(i, j) должна упорядочить элементы $A[i]$, ..., $A[j]$. Предварительный набросок процедуры показан в листинге 8.5. Отметим, что если все элементы $A[i]$, ..., $A[j]$ имеют одинаковые ключи, над ними не производятся никакие действия.

Листинг 8.5. Процедура быстрой сортировки

```
(1)     if A[i], ..., A[j] имеют не менее двух различных ключей then begin
(2)         пусть v — наибольший из первых двух найденных различных
                ключей;
(3)         переставляются элементы A[i], ..., A[j] так, чтобы
                для некоторого k, i+1≤k≤j, A[i], ..., A[k-1] имели ключи,
                меньшие, чем v, а A[k], ..., A[j] — большие или равные v;
(4)         quicksort(i, k-1);
(5)         quicksort(k, j)
        end
```

Напишем функцию *findpivot* (нахождение опорного элемента), реализующую проверку в строке (1) листинга 8.5, т.е. проверяющую, все ли элементы $A[i]$, ..., $A[j]$ одинаковы. Если функция *findpivot* не находит различных ключей, то возвращает

---

[1] Строго говоря, описываемый далее алгоритм быстрой сортировки имеет время выполнения $O(n \log n)$ только в среднем, в самом худшем случае его временная сложность имеет порядок $O(n^2)$.

[2] Интересно отметить, что название *quicksort* (быстрая сортировка) этому алгоритму дал его автор Хоар (Hoare C. A. R.) в работе [49] (интересная статья, в которой исчерпывающе описан данный алгоритм), и в дальнейшем никто не оспаривал это название. — *Прим. ред.*

значение 0. В противном случае она возвращает индекс наибольшего из первых двух различных ключей. Этот наибольший ключ становится опорным элементом. Код функции *findpivot* приведен в листинге 8.6.

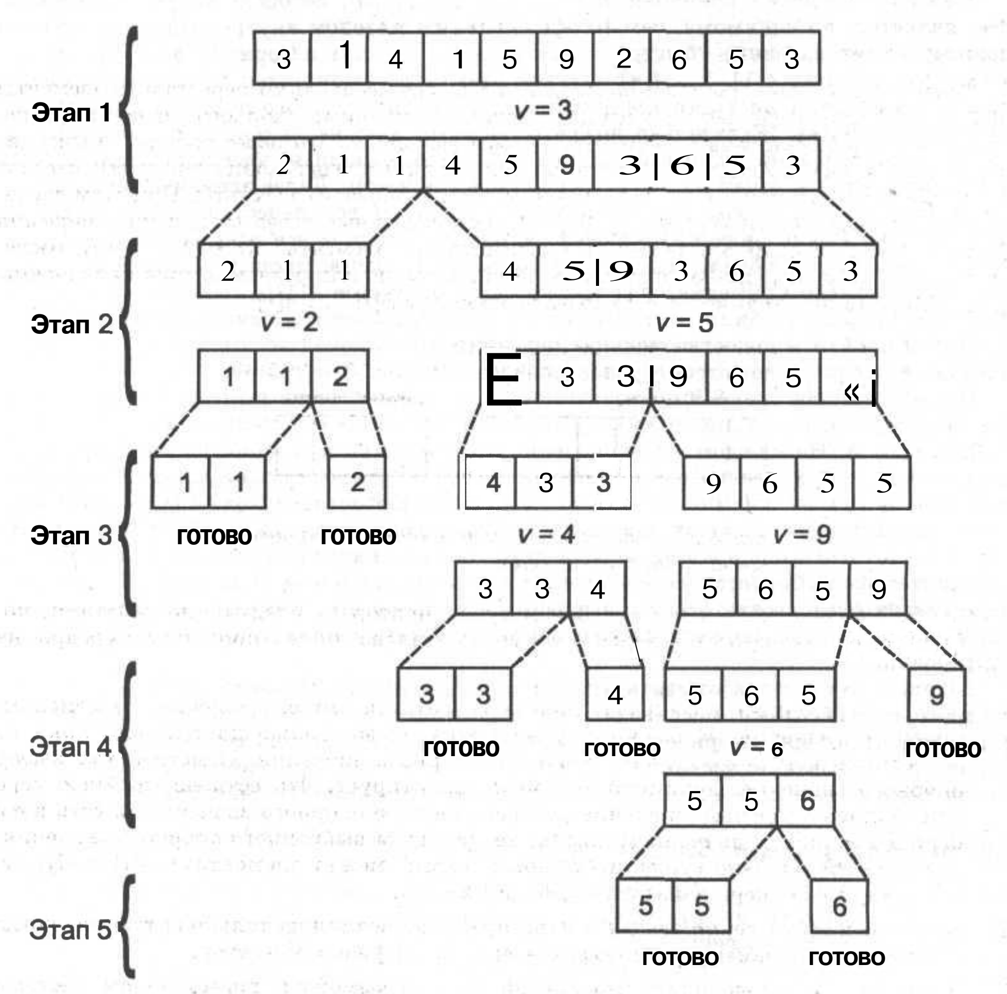


Рис. 8.1. Этапы быстрой сортировки

Листинг 8.6. Функция *findpivot*

```
function findpivot ( i, j: integer ): integer;
    var
        firstkey: keytype;
            { примет значение первого найденного ключа,
              т.е. A[i].key }
        k: integer; { текущий индекс при поиске различных ключей}
    begin
        firstkey:= A[i] .key;
        for k:= i + 1 to j do { просмотр ключей }
            if A[k].key > firstkey then { выбор наибольшего ключа}
```

```
            return(k)
        else if A[k].key < firstkey then
            return(i);
    return(0) { различные ключи не найдены }
end; { findpivot }
```

Теперь реализуем строку (3) из листинга 8.5, где необходимо переставить элементы $A[i]$, ..., $A[j]$ так, чтобы все элементы с ключами, меньшими опорного значения, располагались слева от остальных **элементов**[1]. Чтобы выполнить эти перестановки, введем два курсора l и r, указывающие соответственно на левый и правый концы той части массива *A,* где в настоящий момент мы переставляем (упорядочиваем) элементы. При этом считаем, что уже все элементы $A[i]$, ..., $A[l-1]$, расположенные слева от l, имеют значения ключей, меньшие опорного значения. Соответственно элементы $A[r+1]$, ..., $A[j]$, расположенные справа от r, имеют значения ключей, большие или равные опорному значению (рис. 8.2). Нам необходимо рассортировать элементы $A[l]$, ..., $A[r]$.

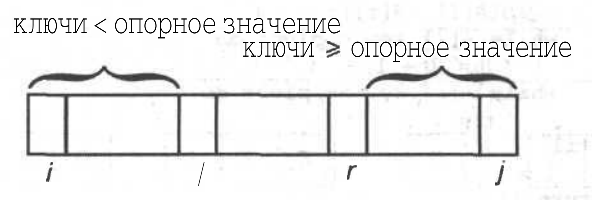


Рис. 8.2. Ситуация, возникающая в процессе перемещения элементов

Сначала положим $l = i$ и $r = j$. Затем будем повторять следующие действия, которые перемещают курсор l вправо, а курсор r влево до тех пор, пока курсоры не пересекутся.

1. Курсор перемещается l вправо, пока не встретится запись с ключом, не меньшим опорного значения. Курсор r перемещается влево, также до тех пор, пока не встретится запись с ключом, меньшим опорного значения. Отметим, что выбор опорного значения функцией *findpivot* гарантирует, что есть по крайней мере одна запись с ключом, значение которого меньше опорного значения, и есть хотя бы одна запись со значением ключа, не меньшим выбранного опорного значения. Поэтому обязательно существует "промежуток" между элементами $A[i]$ и $A[j]$, по которому могут перемещаться курсоры l и r.
2. Выполняется проверка: если $l > r$ (на практике возможна только ситуация, когда $l = r + 1$), то перемещение элементов $A[i]$, ..., $A[j]$ заканчивается.
3. В случае $l < r$ (очевидно, что случай $l = r$ невозможен) переставляем местами элементы $A[l]$ и $A[r]$. После этого запись $A[l]$ будет иметь значение ключа меньшее, чем опорное значение, а $A[r]$ — большее или равное опорному значению. Курсор l перемещается на одну позицию от предыдущего положения вправо, а курсор r — на одну позицию влево. Далее процесс продолжается с пункта 1.

Описанный циклический процесс несколько неуклюжий, поскольку проверка, приводящая к окончанию процесса, расположена посредине. Если этот процесс оформить в виде цикла repeat, то этап 3 надо переместить в начало. Но в этом случае при $l = i$ и $r = j$ сразу меняются местами элементы $A[i]$ и $A[j]$, независимо от того, надо это делать или нет. Но с другой стороны, это несущественно, так как мы не предполагаем перво-

[1] Можно скопировать элементы $A[i]$, ..., $A[j]$ в другой массив, упорядочить **их там,** а затем вернуть обратно в ячейки $A[i]$ . . . $A[j]$ массива *A.* Но мы не будем обсуждать этот подход, так как он требует дополнительной памяти и удлиняет сам процесс сортировки по сравнению с рассматриваемым подходом, когда перестановки элементов выполняются непосредственно в массиве А.

начальное упорядочивание элементов *A[i]*, ..., *A[j]*. В любом случае, читатель должен знать о таких "шалостях" алгоритма, поэтому они не должны его озадачивать. Функция *partition* (разделение), которая выполняет перестановки элементов и возвращает индекс *l*, указывающий на точку разделения данного фрагмента массива А на основе заданного опорного значения *pivot*, приведена в листинге 8.7.

Листинг 8.7. Функция *partition*

```
    function partition ( i, j: integer; pivot: keytype ): integer;
        var
            l, r: integer; { курсоры }
        begin
(1)         l:= i;
(2)         r:= j;
            repeat
(3)             swap(A[l], A[r]);
(4)             while A[l].key < pivot do
(5)                 l:= l + 1;
(6)             while A[r] .key >= pivot do
(7)                 r:= r - 1
            until
(8)             l > r;
(9)         return(l)
        end; { partition }
```

Теперь можно преобразовать эскиз алгоритма быстрой сортировки (листинг 8.5) в настоящую программу *quicksort* (быстрая сортировка). Код этой программы приведен в листинге 8.8. Для сортировки элементов массива *A* типа **array[1..*n*]** of recordtype надо просто вызвать процедуру ***quicksort*(1, *n*)**.

Листинг 8.8. Процедура быстрой сортировки

```
    procedure quicksort ( i, j: integer );
        { сортирует элементы A[i], ..., A[j] внешнего массива A }
        var
            pivot: keytype; { опорное значение }
            pivotindex: integer;
                { индекс элемента массива A, чей ключ равен pivot }
            k: integer;
                {начальный индекс группы элементов, чьи ключи > pivot}
        begin
(1)         pivotindex:= findpivot(i, j);
(2)         if pivotindex <> 0 then begin
                { если все ключи равны, то ничего делать не надо }
(3)             pivot:= A[pivotindex].key;
(4)             k:= partition(i, j, pivot);
(5)             quicksort(i, k-1);
(6)             quicksort(k, j)
            end
        end; { quicksort }
```

Временная сложность быстрой сортировки

Покажем, что время выполнения быстрой сортировки n элементов составляет в среднем $O(n \log n)$ и $O(n^2)$ в худшем случае. В качестве первого шага в доказательстве обоих утверждений надо показать, что время выполнения процедуры *partition*(*i*, *j*, *v*)

имеет порядок $O(j - i + 1)$, т.е. пропорционально числу элементов, над которыми она выполняется.

Чтобы "разобраться" с временем выполнения процедуры *partition,* воспользуемся приемом, который часто применяется при анализе алгоритмов. Надо найти определенные "элементы", для которых в принципе можно вычислить время выполнения, а затем необходимо показать, что каждый шаг алгоритма для "обработки" одного "элемента" требует времени, не превышающего **константы.** Тогда общее время выполнения алгоритма можно вычислить как произведение этой константы на количество "элементов".

В данном случае в качестве "элементов" можно просто взять элементы $A[i]$, ..., $A[j]$ и для каждого из них оценить время, необходимое для достижения в процедуре *partition* курсорами l и r этого элемента, начиная от исходного положения этих курсоров. Сначала отметим, что эта процедура всегда возвращает значение курсора, разбивающее множество элементов $A[i]$, ..., $A[j]$ на две группы, причем каждая из этих групп содержит не менее одного элемента. Далее, так как процедура заканчивается только тогда, когда курсор l "перейдет" через курсор r, то отсюда следует, что каждый элемент просматривается процедурой хотя бы один раз.

Перемещение курсоров осуществляется в циклах строк (4) и (6) листинга 8.7 путем увеличения на 1 значения l или уменьшения на 1 значения r. Сколько шагов может сделать алгоритм между двумя выполнениями оператора $l := l + 1$ или оператора $r := r - 1$? Для того чтобы смоделировать самый худший случай, надо обратиться к началу процедуры. В строках (1), (2) происходит инициализация курсоров l и r. Затем обязательно выполняется перестановка элементов в строке (3). Далее предположим, что циклы строк (4) и (6) "перескакивают" без изменения значений курсоров l и r. На втором и последующих итерациях цикла repeat перестановка в строке (3) гарантирует, что хотя бы один из циклов while строк (4) и (6) будет выполнен не менее одного раза. Следовательно, между выполнениями оператора $l := l + 1$ или $r := r - 1$ в самом худшем случае выполняются строки (1), (2), дважды строка (3) и проверки логических условий в строках (4), (6), (8) и снова в строке (4). Все эти операции требуют фиксированного времени, независящего от значений i и j.

В конце исполнения процедуры выполняются проверки в строках (4), (6) и (8), не связанные ни с какими "элементами", но время этих операций также фиксировано и поэтому его можно "присоединить" к времени обработки какого-либо элемента. Из всего вышесказанного следует вывод, что существует такая временная константа с, что на обработку любого элемента затрачивается время, не превышающее *с.* Поскольку процедура **partition**(*i, j, v*) обрабатывает $j - i + 1$ элементов, то, следовательно, общее время выполнения этой процедуры имеет порядок $O(j - i + 1)$.

Вернемся к оценке времени выполнения процедуры **quicksort**(*i, j*). Легко проверить, что вызов процедуры *findpivot* в строке (1) листинга 8.8 занимает время порядка $O(j - i + 1)$, а в большинстве случаев значительно меньше. Проверка логического условия в строке (2) выполняется за фиксированное время, так же, как и оператор в строке (3), если, конечно, он выполняется. Вызов процедуры **partition**(*i, j, pivot*), как мы показали, требует времени порядка $O(j - i + 1)$. Таким образом, без учета рекурсивных вызовов *quicksort* каждый отдельный вызов этой процедуры требует времени, по крайней мере пропорционального количеству элементов, упорядочиваемых этим вызовом *quicksort.*

Если посмотреть с другой стороны, то общее время выполнения процедуры *quicksort* является суммой по всем элементам количества времени, в течение которого элементы находятся в части массива *A,* обрабатываемой данным вызовом процедуры *quicksort.* Вернемся к рис. 8.1, где вызовы процедуры *quicksort* показаны по этапам (уровням). Очевидно, что никакой элемент не может участвовать в двух вызовах процедуры *quicksort* на одном уровне, поэтому время выполнения данной процедуры можно выразить как сумму по всем элементам определенной *глубины,* т.е. максимального уровня, на котором заканчивается сортировка для данного эле-

мента. Например, из рис. 8.1 видно, что элемент 1 является элементом глубины 3, а элемент 6 — глубины 5.

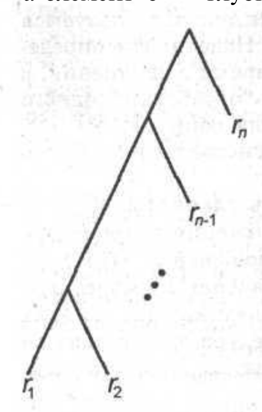


Рис. 8.3. Наихудшая возможная последовательность выбора порны элементов

Чтобы реализовать самый худший случай выполнения процедуры *quicksort*, мы должны управлять выбором опорного элемента, например выбрать в качестве опорного значения наибольшее значение ключа в сортируемом множестве элементов. Тогда сортируемое множество элементов разбивается на два подмножества, одно из которых содержит только один элемент (ключевое значение этого элемента совпадает с опорным значением). Такая последовательность разбиения исходного множества приводит к структуре дерева, подобного изображенному на рис. 8.3, где записи $r_1, r_2 \dots r_n$ заранее упорядочены в порядке возрастания ключей.

В приведенном примере элемент r_i $(2 < i < n)$ имеет глубину $n - i - 1$, а глубина элемента r_1 составляет л - 1. Сумма глубин элементов равна

$$n - 1 + \sum_{i=2}^{n}(n - i + 1) = \frac{n^2}{2} + \frac{n}{2} - 1,$$

т е имеет порядок $\Omega(n^2)$Таким образом, показано, что время выполнения алгоритма быстрой сортировки n элементов в самом худшем случае пропорционально n^2.

Время выполнения быстрой сортировки в среднем

Как всегда, выражение "время выполнения в среднем" понимается как усреднение времени выполнения сортировки по всем возможным упорядочениям исходных наборов элементов в предположении, что все упорядочения равновероятны. Для простоты также предположим, что не существует записей с одинаковыми ключами. В общем случае при возможном равенстве значений некоторых ключей анализ времени выполнения также осуществляется сравнительно просто, но несколько по-иному.

Сделаем еще одно предположение, которое тоже упрощает анализ. Будем считать, что при вызове процедуры *quicksort(i, j)* все упорядочивания элементов $A[i]$, ..., $A[j]$ равновероятны. Это предположение можно попытаться обосновать исходя из того очевидного факта, что опорное значение v, примененное в предыдущем вызове *quicksort,* нельзя использовать для разбиения данного подмножества элементов, так как здесь все элементы имеют значения ключей либо меньшие v, либо большие, либо равные v.[1] Более тщательный анализ программы быстрой Сортировки показывает, что предыдущий опорный элемент скорее всего будет находиться возле правого конца подмножества элементов, равных или больших этого опорного значения (т.е. не будет находиться с равной вероятностью в любом месте этого подмножества), но для больших множеств этот факт не имеет определяющего значения.[2]

Обозначим через $T(n)$ среднее время, затрачиваемое алгоритмом быстрой сортировки на упорядочивание последовательности из n элементов. Очевидно, что время $T(1)$ равно некоторой константе c_1, поскольку имеется только один элемент и не ис-

---

[1] Конечно, порядок элементов $A[i]$, ..., $A[j]$ будет зависеть от первоначального упорядочивания элементов $A[1]$, ..., $A[n]$ и от ранее выбранных опорных элементов (см. в тексте следующее предложение и следующий абзац). Но основываясь на предположении о равных вероятностях всех упорядочений исходной последовательности элементов и на том факте, что предыдущее опорное значение влияет на количество и состав элементов $A[i]$. . . . $A[j]$, но *мало влияет* на их взаимное расположение, в первом приближении гипотеза о равновероятности всех упорядочений элементов $A[i]$, ..., $A[j]$ выглядит вполне приемлемой. — *Прим. ред.*

[2] Из приведенных рассуждений можно сделать практический вывод: если сортировка выполняется не так быстро, как ожидалась, то можно предварительно переупорядочить сортируемые элементы в случайном порядке, а затем повторить быструю сортировку.

пользуется рекурсивный вызов процедуры *quicksort* самой себя. При $n > 1$ требуется время c_2n (c_2 — некоторая константа), чтобы определить опорное значение и разбить исходное множество элементов на два подмножества; после этого вызывается процедура *quicksort* для каждого из этих подмножеств. Было бы хорошо (с точки зрения анализа алгоритма), если бы мы могли утверждать, что для подмножества, которое упорядочивается данным вызовом процедуры *quicksort,* опорное значение с одинаковой вероятностью может быть равным любому из n элементов исходного множества. Но в этом случае мы не можем гарантировать, что такое опорное значение сможет разбить это подмножества на два других непустых подмножества, одно из которых содержало бы элементы, меньшие этого опорного значения, а другое — равные или большие его. Другими словами, в рассматриваемом подмножестве с ненулевой вероятностью могут находиться только элементы, меньшие этого опорного значения, либо, наоборот, большие или равные опорному значению. Именно по этой причине в алгоритме быстрой сортировки опорное значение выбирается как наибольшее значение ключей первых двух различных элементов. Отметим, что такое определение опорного значения не приводит к эффективному (т.е. примерно равному) размеру подмножеств, на которые разобьется рассматриваемое подмножество относительно этого опорного значения. При таком выборе опорного значения можно заметить тенденцию, что "левое" подмножество, состоящее из элементов, чьи ключи меньше опорного значения, будет больше "правого" подмножества, состоящего из элементов, ключи которых больше или равны опорному значению.

Предполагая, что все сортируемые элементы различны, найдем вероятность того, что "левое" подмножество будет содержать i из n элементов. Для того чтобы "левое" подмножество содержало i элементов, необходимо, чтобы опорное значение совпадало с $(i + 1)$-м порядковым значением в последовательности упорядоченных в возрастающем порядке элементов исходного множества. При нашем методе выбора опорного значения данное значение можно получить в двух случаях: (1) если элемент со значением ключа, равного $(i + 1)$-му порядковому значению, стоит в первой позиции, во второй позиции стоит любой из i элементов с ключами, меньшими, чем у первого элемента; (2) элемент со значением ключа, равного новому опорному значению, стоит на второй позиции, а в первой — любой из i элементов с ключами, меньшими чем у второго элемента. Вероятность того, что отдельный элемент, такой как $(i + 1)$-е порядковое значение, появится в первой позиции, равна $1/n$. Вероятность того, что во второй позиции будет стоять любой из i элементов со значением ключа, меньшим, чем у первого элемента, равна $i/(n - 1)$. Поэтому вероятность того, что новое опорное значение находится в первой позиции, равна $i/n(n - 1)$. Такое же значение имеет вероятность события, что новое опорное значение появится во второй позиции. Отсюда вытекает, что "левое" множество будет иметь размер i с вероятность $2i/n(n - 1)$ для $1 < i < n$.

Теперь можно записать рекуррентное соотношение для $T(n)$:

$$T(n) < \sum_{i=1}^{n-1} \frac{2i}{n(n-1)} \left[T(i) + T(n-i)\right] + c_2 n. \tag{8.1}$$

Неравенство (8.1) показывает, что среднее время выполнения алгоритма быстрой сортировки не превышает времени c_2n, прошедшего от начала алгоритма до рекурсивных вызовов *quicksort,* плюс среднее время этих рекурсивных вызовов. Последнее время является суммой по всем возможным i произведений вероятностей того, что "левое" множество состоит из i элементов (или, что то же самое, вероятностей того, что "правое" множество состоит из $n - i$ элементов) и времени рекурсивных вызовов $T(i)$ и $T(n - i)$.

Теперь надо упростить сумму в выражении (8.1). Сначала заметим, что для любой функции $f(i)$ путем замены i на $n - i$ можно доказать следующее равенство:

$$\sum_{1=1}^{n-1} f(i) = \sum_{i=1}^{n-1} f(n-i). \tag{8.2}$$

Из равенства (8.2) следует, что

$$\sum_{i=1}^{n-1} f(i) = \frac{1}{2}\sum_{i=1}^{n-1}(f(i) + f(n-i)). \tag{8.3}$$

Положив $f(i)$ равными слагаемым в выражении (8.1) и применяя равенство (8.3), из неравенства (8.1) получим

$$T(n) \le \frac{1}{2}\sum_{i=1}^{n-1}\left\{\frac{2i}{n(n-1)}[T(i)+T(n-i)] + \frac{2(n-i)}{n(n-1)}[T(n-i)+T(i)]\right\} + c_2 n \le$$

$$\le \frac{1}{n-1}\sum_{i=1}^{n-1}[T(i) + T(n-i)] + c_2 n. \tag{8.4}$$

Далее снова применяем формулу (8.3) к последней сумме в (8.4), положив $f(i) = T(i)$:

$$T(n) \le \frac{2}{n-1}\sum_{i=1}^{n-1} T(i) + c_2 n. \tag{8.5}$$

(Отметим, что последнее неравенство в (8.4) соответствует выражению, которое мы получили бы, если бы предполагали равные вероятности для всех размеров (от 1 до n) "левых" множеств.) Решение рекуррентных соотношений мы подробно рассмотрим в главе 9. Здесь мы покажем, как можно оценить сверху решение рекуррентного неравенства (8.5). Мы докажем, что для всех $n \ge 2$ $T(n) < cn\log n$ для некоторой константы c.

Доказательство проведем методом индукции по n. Для $n = 2$ непосредственно из неравенства (8.5) для некоторой константы с получаем $T(2) \le 2c - 2c\log 2$. Далее, в соответствии с методом математической индукции, предположим, что для $i < n$ выполняются неравенства $T(i) < ci\log i$. Подставив эти неравенства в формулу (8.5), для $T(n)$ получим

$$T(n) \le \frac{2c}{n-1}\sum_{i=1}^{n-1} i\log i + c_2 n. \tag{8.6}$$

Разобьем сумму в (8.6) на две суммы: в первой $i \le n/2$, во второй $i > n/2$. В первой сумме $\log i < \log(n/2) = \log n - 1$, во второй сумме $\log i$ не превышают $\log$ п. Таким образом, из (8.6) получаем

$$T(n) \le \frac{2c}{n-1}\left[\sum_{i=1}^{n/2} i\log i + \sum_{i=n/2+1}^{n-1} i\log i\right] + c_2 n \le$$

$$\le \frac{2c}{n-1}\left[\sum_{i=1}^{n/2} i(\log n - 1) + \sum_{i=n/2+1}^{n-1} i\log n\right] + c_2 n \le$$

$$\le \frac{2c}{n-1}\left[\frac{n}{4}\left(\frac{n}{2}+1\right)\log n - \frac{n}{4}\left(\frac{n}{2}+1\right) + \frac{3}{4}n\left(\frac{n}{2}-1\right)\log n\right] + c_2 n =$$

$$= \frac{2c}{n-1}\left[\left(\frac{n^2}{2} - \frac{n}{2}\right)\log n - \left(\frac{n^2}{8} + \frac{n}{4}\right)\right] + c_2 n \le$$

$$\le cn\log n - \frac{cn}{4} - \frac{cn}{2(n-1)} + c_2 n. \tag{8.7}$$

Если положить $c > 4c_2$, тогда в последнем выражении сумма второго и четвертого слагаемых не превысит нуля. Третье слагаемое в последней сумме (8.7) отрицательное, поэтому можно утверждать, что $T(n) \le cn\log n$ для константы $c = 4c_2$. Этим заканчивается доказательство того, среднее время выполнения алгоритма быстрой сортировки составляет $O(n\log n)$.

Реализация алгоритма быстрой сортировки

Алгоритм быстрой сортировки можно реализовать не только посредством процедура *quicksort* так, чтобы среднее время выполнения равнялось $O(n \log n)$. Можно создать другие процедуры, реализующие этот алгоритм, которые будут иметь тот же порядок $O(n \log n)$ времени выполнения в среднем, но за счет меньшей константы пропорциональности в этой оценке будут работать несколько быстрее (напомним, что порядок времени выполнения, такой как $O(n \log n)$, определяется с точностью до константы пропорциональности). Эту константу можно уменьшить, если выбирать опорное значение таким, чтобы оно разбивало рассматриваемое в данный момент подмножество элементов на две примерно равные части. Если всегда такие подмножества разбиваются точно пополам, тогда каждый сортируемый элемент имеет глубину точно logn в дереве, аналогичном изображению из рис. 8.1 (вспомните полные двоичные деревья из главы 5). Для сравнения: средняя глубина элементов в процедуре *quicksort* (листинг 8.8) составляет 1.4 logn. Так что можно надеяться, что при "правильном" выборе опорного значения выполнение алгоритма ускорится.

Например, можно с помощью датчика случайных чисел выбрать три элемента из подмножества и средний (по величине) из них назначить опорным значением. Можно также, задав некоторое число k, выбрать случайным образом (например, опять с помощью датчика случайных чисел) k элементов из подмножества, упорядочить их процедурой *quicksort* или посредством какой-либо простой сортировки из раздела 8.2 и в качестве опорного значения взять медиану (т.е. $(k + 1)/2$-й элемент) этих k элементов.[1] Заметим в скобках, что интересной задачей является определение наилучшего значения k как функции от количества элементов в подмножестве, подлежащем сортировке. Если k очень мало, то среднее время будет близко к тому, как если бы выбирался только один элемент. Если же k очень велико, то уже необходимо учитывать время нахождения медианы среди k элементов.

Другие реализации алгоритма быстрой сортировки можно получить в зависимости от того, что мы делаем в случае, когда получаются малые подмножества. Напомним, что в разделе 8.2 говорилось о том, что при малых п алгоритмы с временем выполнения $O(n^2)$ работают быстрее, чем алгоритмы с временем выполнения порядка $O(n \log n)$. Однако понятие "малости" п зависит от многих факторов, таких как организация рекурсивных вызовов, машинная архитектура и компилятор языка программирования, на котором написана процедура сортировки. В книге [65] предлагается значение 9 в качестве порогового значения размера подмножества, ниже которого целесообразно применять простые методы сортировки.

Существует еще один метод "ускорения" быстрой сортировки за счет увеличения используемого пространства памяти компьютера. Отметим, что этот метод применим к любым алгоритмам сортировки. Если есть достаточный объем свободной памяти, то можно создать массив указателей на записи массива *A*. Затем следует организовать процедуру сравнения ключей записей посредством этих указателей. В этом случае нет необходимости в физическом перемещении записей, достаточно перемещать указатели, что значительно быстрее перемещения непосредственно записей, особенно когда они большие (состоят из многих полей). В конце выполнения процедуры сортировки указатели будут отсортированы слева направо, указывая на записи в нужном порядке. Затем сравнительно просто можно переставить сами записи в правильном порядке.

Таким образом, можно сделать только п перестановок записей вместо $O(n \log n)$ перестановок, и эта разница особенно значительна для больших записей. Отрицательными аспектами такого подхода являются использование дополнительного объема памяти для массива указателей и более медленное выполнение операции сравнения ключей, поскольку здесь сначала, следуя за указателями, надо найти нужные записи, затем необходимо войти в записи и только потом взять значения поля ключа.

---

[1] Поскольку в данном случае необходимо только значение медианы, то вместо упорядочивания всего списка из k элементов рациональнее применить один из быстрых алгоритмов нахождения значения медианы, описанных в разделе 8.7.

8.4. Пирамидальная сортировка

В этом разделе мы рассмотрим алгоритм сортировки, называемой *пирамидальной*[1], его время выполнения в худшем случае такое, как и в среднем, и имеет порядок $O(n \log n)$. Этот алгоритм можно записать в абстрактной (обобщенной) форме, используя операторы множеств INSERT, DELETE, EMPTY и MIN, введенные в главах 4 и 5. Обозначим через L список элементов, подлежащих сортировке, а S — множество элементов типа recordtype (тип записи), которое будет использоваться для хранения сортируемых элементов. Оператор MIN применяется к ключевому полю записей, т.е. MIN(S) возвращает запись из множества S с минимальным значением ключа. В листинге 8.9 показан абстрактный алгоритм сортировки, который мы далее преобразуем в алгоритм пирамидальной сортировки.

Листинг 8.9. Абстрактный **алгоритм сортировки**

```
(1)      for x e L do
(2)          INSERT(x, S) ;
(3)      while not EMPTY(S) do begin
(4)          y:= MIN(S) ;
(5)          writeln(y) ;
(6)          DELETE(y, S)
         end
```

В главах 4 и 5 мы показали различные структуры данных, такие как 2-3 деревья, помогающие реализовать эти операторы множеств с временем выполнения $O(\log n)$. Если список L содержит n элементов, то наш абстрактный алгоритм требует выполнения n операторов INSERT, n операторов MIN, n операторов DELETE и $n + 1$ операторов EMPTY. Таким образом, общее время выполнения алгоритма имеет порядок $O(n \log n)$, если, конечно, используется подходящая структура данных.

Структура данных частично упорядоченных деревьев из раздела 4.11 хорошо подходит для данного алгоритма. Напомним, что частично упорядоченное дерево можно представить в виде *кучи* — массива A, чьи элементы подчиняются неравенствам $A[i].key < A[2*i].key$ и $A[i].key < A[2*i + 1].key$. Если интерпретировать элементы с индексами $2i$ и $2i + 1$ как "сыновей" элемента с индексом i, тогда массив A формирует сбалансированное двоичное дерево, в котором значения ключей родителей никогда не превосходят значения ключей сыновей.

В разделе 4.11 было показано, что частично упорядоченное дерево поддерживает операторы INSERT и DELETEMIN с временем выполнения $O(\log n)$. Но на частично упорядоченном дереве нельзя реализовать общий оператор DELETE с временем выполнения $O(\log n)$ (всегда найдется отдельный элемент, требующий линейного времени удаления в самом худшем случае). В связи с этим отметим, что в листинге 8.9 удаляются только элементы, имеющие минимальные значения ключей. Поэтому строки (4) и (6) можно объединить одним оператором DELETEMIN, который будет возвращать элемент y. Таким образом, используя структуру данных частично упорядоченного дерева из раздела 4.11, можно выполнить абстрактный алгоритм сортировки за время $O(n \log n)$.

Сделаем еще одно изменение в алгоритме листинга 8.9, чтобы избежать печати удаляемых элементов. Заметим, что множество S всегда хранится в виде кучи в верхней части массива A, даже когда содержит только i элементов ($i < n$). Для частично упорядоченного дерева наименьший элемент всегда хранится в ячейке $A[1]$. Элементы, уже удаленные из множества S, можно было бы хранить в ячейках

[1] Свое название этот алгоритм получил вследствие того, что применяемое здесь частично упорядоченное сортирующее дерево имеет вид "пирамиды". Отметим также, что в русском издании книги [3] этот метод сортировки назван "сортдеревом", т.е. упорядочиванием посредством сортирующего дерева. — *Прим. ред.*

$A[i + 1]$, ..., $A[n]$, сортированными в обратном порядке, т.е. так, чтобы выполнялись неравенства $A[i + 1] > A[i + 2] > \ldots \geq A[n]$.[1] Поскольку элемент $A[1]$ является наименьшим среди элементов $A[1]$, ..., $A[i]$, то для повышения эффективности оператора DELETEMIN можно просто поменять местами элементы $A[1]$ и $A[i]$. Поскольку новый элемент $A[i]$ (т.е. старый $A[1]$) не меньше, чем $A[i + 1]$ (элемент $A[i + 1]$ был удален из множества S на предыдущем шаге как наименьший), то получим последовательность $A[i]$, ..., $A[n]$, отсортированными в убывающем порядке. Теперь надо заняться размещением элементов $A[1]$, ..., $A[i - 1]$.

Поскольку новый элемент $A[1]$ (старый элемент $A[i]$) нарушает структуру частично упорядоченного дерева, его надо "протолкнуть" вниз по ветвям дерева, как это делается в процедуре DELETEMIN листинга 4.13. Здесь для этих целей мы используем процедуру *pushdown* (протолкнуть вниз), оперирующую с массивом *A*, определенным вне этой процедуры. Код процедуры *pushdown* приведен в листинге 8.10. С помощью последовательности перестановок данная процедура "проталкивает" элемент *A*[*first*] вниз по дереву до нужной позиции. Чтобы восстановить частично упорядоченное дерево в случае нашего алгоритма сортировки, надо вызвать процедуру *pushdown* для *first* — 1.

Листинг 8.10. Процедура *pushdown*

```
procedure pushdown ( first, last: integer );
    { Элементы A[first], ..., A[last] составляют частично
      упорядоченное дерево за исключением, возможно, элемента
      A[first] и его сыновей. Процедура pushdown
      восстанавливает частично упорядоченное дерево }
  var
      r: integer; { указывает текущую позицию A[first] }
  begin
      r:= first; { инициализация }
      while r <= last div 2 do
          if last = 2*r then begin
              { элемент в позиции r имеет одного сына
                  в позиции 2*r }
              if A[r].key > A[2*r].key then
                  swap(A[r], A[2*r]);
                  r:= last { досрочный выход из цикла while }
              end
          else { элемент в позиции r имеет двух сыновей
                  в позициях 2*r и 2*r + 1 }
              if A[r].key > A[2*r].key and
                  A[2*r].key <= A[2*r + 1].key then begin
                      { перестановка элемента в позиции r
                          с левым сыном }
                      swap(A[r], A[2*r]);
                      r:= 2*r
              end
              else if A[r].key > A[2*r + 1].key and
                  A[2*r + 1].key < A[2*r].key then begin
                      { перестановка элемента в позиции r
                          с правым сыном }
```

[1] В таком случае в конце выполнения алгоритма сортировки мы получим массив *A*, содержащий элементы, упорядоченные в обратном порядке. Если необходимо, чтобы в конце сортировки массив A содержал элементы в возрастающем порядке (начиная с наименьшего), то надо просто заменить оператор DELETEMIN на оператор DELETEMAX, а частично упорядоченное дерево организовать так, чтобы каждый родитель имел значение ключа не меньшее, чем у его сыновей.

```
                        swap(A[r], A[2*r + 1]);
                        r:= 2*r + 1
                    end
                    else { элемент в позиции r не нарушает порядок
                           в частично упорядоченном дереве }
                        r:= last { выход из цикла while }.
end; { pushdown }
```

Вернемся к листингу 8.9 и займемся строками (4) - (6). Выбор минимального элемента в строке (4) прост — это всегда элемент *A*[1]. Чтобы исключить оператор печати в строке (5), мы поменяли местами элементы *A*[1] и *A*[*i*] в текущей куче. Удалить минимальный элемент из текущей кучи также легко: надо просто уменьшить на единицу курсор i, указывающий на конец текущей кучи. Затем надо вызвать процедуру ***pushdown*(1, *i* -** 1) для восстановления порядка в частично упорядоченном дереве кучи *A*[1], ..., *A*[*i* − 1].

Вместо проверки в строке (3), не является ли множество S пустым, можно проверить значение курсора *i,* указывающего на конец текущей кучи. Теперь осталось рассмотреть способ выполнения операторов в строках (1), (2). Можно с самого начала поместить элементы списка *L* в массив *A* в произвольном порядке. Для создания первоначального частично упорядоченного дерева надо последовательно вызывать процедуру ***pushdown*(*j*, *n*)** для всех *j* = л/2, л/2 - 1, ..., **1.** Легко видеть, что после вызова процедуры ***pushdown*(*j*, *n*)** порядок в ранее упорядоченной части строящегося дерева не нарушается, так как новый элемент, добавляемый в дерево, не вносит нового нарушения порядка, поскольку он только меняется местами со своим "меньшим" сыном. Полный код процедуры ***heapsort*** (пирамидальная сортировка) показан в листинге 8.11.

Листинг 8.11. Процедура пирамидальной сортировки

```
    procedure heapsort;
        { Сортирует элементы массива A[1], ..., A[n]
            в убывающем порядке }
        var
            i: integer; { курсор в массиве A }
        begin
            { создание частично упорядоченного дерева }
(1)         for i:= n div 2 downto 1 do
(2)             pushdown(i, n);
(3)         for i:= n downto 2 do begin
(4)             swap(A[1], A[i]);
                    { удаление минимального элемента из кучи }
(5)             pushdown(1, i - 1)
                    { восстановление частично упорядоченного дерева }
            end
        end; { heapsort }
```

Анализ пирамидальной сортировки

Сначала оценим время выполнения процедуры *pushdown.* Из листинга 8.10 видно, что тело цикла while (т.е. отдельная итерация этого цикла) выполняется за фиксированное время. После каждой итерации переменная *r* по крайней мере вдвое увеличивает свое значение. Поэтому, учитывая, что начальное значение *r* равно *first,* после *i* итераций будем иметь $r > first * 2^i$. Цикл while выполняется до тех пор, пока $r > last/2$. Это условие будет выполняться, если выполняется неравенство $first * 2^i > last/2$. Последнее неравенство можно переписать как

$$i > \log(last/first) - 1. \quad (8.8)$$

Следовательно, число итераций цикл while в процедуре *pushdown* не превышает $\log(last/first)$.

Поскольку $first > 1$ и $last < n$, то из (8.8) следует, что на каждый вызов процедуры *pushdown* в строках (2) и (5) листинга 8.11 затрачивалось время, по порядку не большее, чем $O(\log n)$. Очевидно, что цикл **for** строк (1), (2) выполняется $n/2$ раз. Поэтому общее время выполнения этого цикла имеет порядок $O(n \log n)$.[1] Цикл в строках (3) - (б) выполняется n - 1 раз. Следовательно, на выполнение всех перестановок в строке (4) тратится время порядка $O(n)$, а на восстановление частично упорядоченного дерева (строка (5)) — $O(n \log n)$. Отсюда вытекает, что общее время выполнения цикла в строках (3) - (б) имеет порядок $O(n \log n)$ и такой же порядок времени выполнения всей процедуры *heapsort*.

Несмотря на то что процедура *heapsort* имеет время выполнения порядка $O(n \log n)$ в самом худшем случае, в среднем ее время несколько хуже, чем в быстрой сортировке, хотя и имеет тот же порядок. Пирамидальная сортировка интересна в теоретическом плане, поскольку это первый рассмотренный нами алгоритм, имеющий в самом худшем случае время выполнения $O(n \log n)$. В практических ситуациях этот алгоритм полезен тогда, когда надо не сортировать все n элементов списка, а только отобрать k наименьших элементов, и при этом k значительно меньше n. В последнем примечании указывалось, что в действительности время выполнения цикла в строках (1), (2) имеет порядок $O(n)$. Если надо сделать только k итераций цикла (3) - (5), то на это потратится время порядка $O(k \log n)$. Поэтому отбор k минимальных элементов процедура *heapsort* выполнит за время порядка $O(n + k \log n)$. Отсюда следует, что при $k < n/\log n$ на выполнение этой операции потребуется время порядка $O(n)$.

8.5. „Карманная" сортировка

Правомерен вопрос: всегда ли при сортировке n элементов нижняя граница времени выполнения имеет порядок $\Omega(n \log n)$? В следующем разделе мы рассмотрим алгоритмы сортировки и их нижнюю границу времени выполнения, если о типе данных ключей не предполагается ничего, кроме того, что их можно упорядочить посредством некой функции, показывающей, когда один ключ "меньше чем" другой. Часто можно получить время сортировки меньшее, чем $O(n \log n)$, но необходима дополнительная информация о сортируемых ключах.

Пример 8.5. Предположим, что значения ключей являются целыми числами из интервала от 1 до п, они не повторяются и число сортируемых элементов также равно п. Если обозначить через *А и В* массивы типа array[1..п] of recordtype, n элементов, подлежащих сортировке, первоначально находятся в массиве *А*, тогда можно организовать поочередное помещение в массив *В* записей в порядке возрастания значений ключей следующим образом:

```
for i:= 1 to n do
    B[A[i].key]:= A[i];                                    (8.9)
```

Этот код вычисляет, где в массиве *В* должен находиться элемент $A[i]$, и помещает его туда. Весь этот цикл требует времени порядка $O(n)$ и работает корректно только тогда, когда значения всех ключей различны и являются целыми числами из интервала от 1 до п.

---

[1] Более точный анализ показывает, что это время имеет порядок $O(n)$. Для i из интервала от п/2 до п/4 + 1 из (8.8) следует, что в процедуре *pushdown* цикл while выполняется не более одного раза. Для i из интервала от п/4 до п/8 + 1 выполняются не более двух итераций и т.д. Поэтому общее количество итераций этого цикла для i из интервала от $n/2$ до 1 ограничено величиной $1*n/4 + 2*n/8 + 3*n/16 + \ldots$. С другой стороны, эта улучшенная оценка не влияет на время выполнения процедуры *heapsort* в целом, так как здесь основное время тратится на выполнение строк (3) - (5).

Существует другой способ сортировки элементов массива A с временем $O(n)$, но без использования второго массива B. Поочередно посетим элементы $A[1]$, ..., $A[n]$. Если запись в ячейке $A[i]$ имеет ключ j и $j \neq i$, то меняются местами записи в ячейках $A[i]$ и $A[j]$. Если после этой перестановки новая запись в ячейке $A[i]$ имеет ключ k и $k \neq i$, то осуществляется перестановка между $A[i]$ и $A[k]$ и т.д. Каждая перестановка помещает хотя бы одну запись в нужном порядке. Поэтому данный алгоритм сортировки элементов массива A на месте имеет время выполнения порядка $O(n)$.

```
for i:= 1 to n do
    while A[i].key <> i do
        swap(A[i], A[A[i].key]);
```

□

Программа (8.9) — это простой пример "карманной" сортировки, где создаются "карманы" для хранения записей с определенным значением ключа. Мы проверяем, имеет ли данная запись ключ со значением r, и если это так, то помещаем эту запись в "карман" для записей, чьи значения ключей равны r. В программе (8.9) "карманами" являются элементы массива B, где $B[i]$ — "карман" для записей с ключевым значением i. Массив в качестве "карманов" можно использовать только в простейшем случае, когда мы знаем, что не может быть более одной записи в одном "кармане". Более того, при использовании массива в качестве "карманов" не возникает необходимости в упорядочивании элементов внутри "кармана", так как в одном "кармане" содержится не более одной записи, а алгоритм построен так, чтобы элементы в массиве располагались в правильном порядке.

Но в общем случае мы должны быть готовы к тому, что в одном "кармане" может храниться несколько записей, а также должны уметь объединять содержимое нескольких "карманов" в один, располагая элементы в объединенном "кармане" в правильном порядке. Для определенности далее будем считать, что элементы массива A имеют тип данных recordtype, а ключи записей — тип keytype. Кроме того, примем, но только в этом разделе, что тип данных keytype является перечислимым типом, т.е. таким, как последовательность целых чисел 1, 2, ..., n, или как символьные строки. Обозначим через listtype (тип списка) тип данных, который представляет списки элементов типа recordtype. Тип listtype может быть любым типом списков, описанным в главе 2, но в данном случае нам наиболее подходят связанные списки, поскольку мы будем их наращивать в "карманах" до размеров, которые нельзя предусмотреть заранее. Можно только предвидеть, что общая длина всех списков фиксирована и равна п, поэтому при необходимости массив из n ячеек может обеспечить реализацию списков для всех "карманов".

Необходим также массив B типа **array[keytype]** of listtype. Это будет массив "карманов", хранящих списки (или заголовки списков, если используются связанные списки). Индексы массива B имеют тип данных keytype, так что каждая ячейка этого массива соответствует одному значению ключа. Таким образом, мы уже можем обобщить программу (8.9), поскольку теперь "карманы" имеют достаточную емкость.

Рассмотрим, как можно выполнить объединение "карманов". Формально над списками a_1, a_2, ..., a_i и Ъ\, b_2, ..., b_j надо выполнить операцию *конкатенации* списков, в результате которой получим список a_1, a_2 . . . a_i, b_1, b_2, ..., b_j. Для реализации оператора конкатенации CONCATENATE(L_1, L_2), который заменяет список L_1 объединенным списком L_1L_2, можно использовать любые представления списков, которые мы изучали в главе 2.

Для более эффективного выполнения оператора конкатенации в добавление к заголовкам списков можно использовать указатели на последние элементы списков (или на заголовок списка, если список пустой). Такое нововведение позволит избежать просмотра всех элементов списка для нахождения последнего. На рис. 8.4 пунктирными линиями показаны измененные указатели при конкатенации списков L_1 и L_2 в один общий список L_1. Список L_2 после объединения списков предполагается "уничтоженным", поэтому указатели на его заголовок и конец "обнуляются".

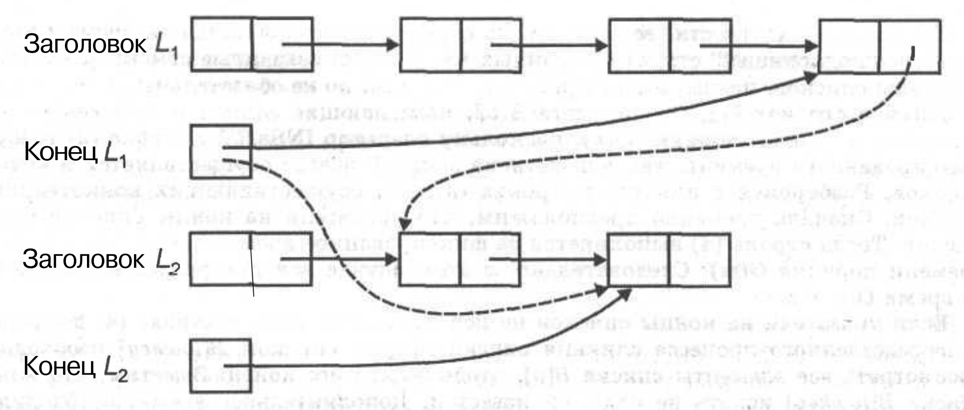


Рис. 8.4. Конкатенация связанных списков

Теперь можно написать программу "карманной" сортировки записей, когда поле ключей является перечислимым типом. Эта программа, показанная в листинге 8.12, написана с использованием базовых операторов, выполняемых над списками. Как уже говорилось, связанные списки являются предпочтительной реализацией "карманов", но, конечно, можно использовать и другие реализации. Напомним также, что в соответствии с нашими предположениями, массив A типа **array[1..*n*]** of recodrtype содержит элементы, подлежащие сортировке, а массив *B* типа **array[keytype]** of listtype предназначен для "карманов". Будем также считать, что переменные перечислительного типа keytype могут изменяться в пределах от ***lowkey*** (нижний ключ) до ***highkey*** (верхний ключ).

Листинг 8.12. Программа *binsort* („карманная" сортировка)[1]

```
    procedure binsort;
        { Сортирует элементы массива A, помещая отсортированный
            список в "карман" B[lowkey] }
        var
            i: integer;
            v: keytype;
        begin
            { начинается занесение записей в "карманы" }
(1)         for i:= 1 to n do
                { помещение элемента A[i] в начало "кармана",
                    соответствующего ключу этого элемента }
(2)             INSERT(A[i], FIRST(B[A[i].key]), B[A[i].key]);
(3)         for v:= succ(lowkey) to highkey do
                {конкатенация всех "карманов" в конец "кармана" B[lowkey]}
(4)             CONCATENATE(B[lowkey], B[v])
        end; { binsort }
```

Анализ „карманной" сортировки

Если сортируется ***n*** элементов, которые имеют ***m*** различных значений ключей (следовательно, *m* различных "карманов"), тогда программа из листинга 8.12 выполняется за время порядка $O(n + m)$, если, конечно, используется подходящая струк-

[1] Для читателя, незнакомого с языком Pascal (или мало знакомого с этим языком), поясним, что применяемая в этом листинге стандартная функция Pascal **succ(*x*)** для данных перечислимого типа возвращает следующее за x значение. — *Прим. ред.*

тура данных. В частности, если $m < n$, то сортировка выполняется за время $O(n)$. В качестве "подходящей" структуры данных мы считаем связанные списки. Указатели на концы списков, показанные на рис. 8.4, полезны, но не обязательны.

Цикл в строках (1), (2) листинга 8.12, помещающие записи в "карманы", выполняется за время порядка $O(n)$, поскольку оператор INSERT в строке (2) требует фиксированного времени, так как вставка записей всегда осуществляется в начало списков. Разберемся с циклом в строках (3), (4), осуществляющих конкатенацию списков. Сначала временно предположим, что указатели на концы списков существуют. Тогда строка (4) выполняется за фиксированное время, а весь цикл требует времени порядка $O(m)$. Следовательно, в этом случае вся программа выполняется за время $O(n + m)$.

Если указатели на концы списков не используются, тогда в строке (4) до начала непосредственного процесса слияния списка $B[v]$ со списком $B[lowkey]$ необходимо просмотреть все элементы списка $B[v]$, чтобы найти его конец. Заметим, что конец списка $B[lowkey]$ искать не надо, он известен. Дополнительное время, необходимое для поиска концов всех "карманов", не превышает $O(n)$, поскольку длина всех "карманов" равна n. Это дополнительное время не влияет на порядок времени выполнения всей процедуры *binsort*, поскольку $O(n)$ не больше, чем $O(n + m)$.

Сортировка множеств с большими значениями ключей

ЕСЛИ количество различных ключей m не превосходит количества элементов n, то время выполнения $O(n + m)$ программы из листинга 8.12 в действительности будет иметь порядок $O(n)$. Но каково будет это время, если $m = n^2$, например? По-видимому, если время выполнения программы $O(n + m)$, то должно быть $O(n + n^2)$, т.е. $O(n^2)$.[1] Но можно ли учесть тот факт, что множество значений ключей ограничено, и улучшить время выполнения программы? Ответ положительный: приведенный ниже алгоритм, являющийся обобщением "карманной" сортировки, даже на множестве значений ключей вида 1, 2, ..., n^k для некоторого фиксированного k выполняет сортировку элементов за время $O(n)$.

Пример 8.6. Рассмотрим задачу сортировки n целых чисел из интервала от 0 до n^2 - 1. Выполним сортировку этих чисел в два этапа. На первом этапе используем n "карманов", индексированных целыми числами от 0 до n - 1. Поместим каждое сортируемое число i в список "кармана" с номером $i \bmod n$. Но в отличие от процедуры листинга 8.12 каждый новый элемент помещается не в начало списка, а в его конец, и это очень важное отличие. Заметим, что если мы хотим эффективно выполнить вставку элементов в конец списков, то надо для представления списков использовать связанные списки, а также указатели на концы списков.

Например, пусть $n = 10$, а список сортируемых чисел состоит из точных квадратов от 0^2 до 9^2, расположенных в случайном порядке 36, 9, 0, 25, 1, 49, 64, 16, 81, 4. В этом случае номер "кармана" для числа i равен самой правой цифре в десятичной записи числа i. В табл. 8.5,а показано размещение сортируемых чисел по спискам "карманов". Отметим, что числа появляются в списках в том же порядке, в каком они расположены в исходном списке, например в списке "кармана" 6 будут располагаться числа 36, 16, а не 16, 36, поскольку в исходном списке число 36 предшествует числу 16.

---

[1] Так как по ранее сделанным предположениям значения ключей имеют перечислимый тип данных, а в языке Pascal всегда можно объявить собственный перечислимый тип, то в данном случае можно объявить тип данных, содержащий все значения ключей. После этого программа из листинга 8.12 будет выполняться за время $O(n + m)$. Но такой подход имеет очевидные недостатки: во-первых, надо перечислить все значения ключей (если их несколько сотен, то это может породить определенные проблемы), во-вторых, значения ключей должны быть записаны в возрастающем порядке, т.е. необходимо провести предварительную сортировку ключей, и, в третьих, программа с таким объявлением типа будет работать только на данном множестве ключей. Именно эти соображения приводят к необходимости модификации алгоритма "карманной" сортировки. — *Прим. ред.*

Теперь содержимое "карманов" поочередно объединим в один список:

$$0, 1, 81, 64, 4, 25, 36, 16, 9, 49. \qquad (8.10)$$

Если в качестве структуры данных мы использовали связанные списки, а также указатели на концы списков, то операция конкатенации n элементов, содержащихся в карманах, займет время порядка $O(n)$.

Целые числа из списка, созданного путем конкатенации на предыдущем этапе, снова распределяются по карманам, но уже по другому принципу. Теперь число i помещается в карман с номером $[i/n]$, т.е. номер кармана равен наибольшему целому числу, равному или меньшему i/n (другими словами, квадратные скобки здесь и далее в этой главе обозначают операцию взятия целой части числа). На этом этапе добавляемые в "карманы" числа также вставляются в концы списков. После распределения списков по "карманам" снова выполняется операция конкатенации всех списков в один список. В результате получим отсортированный список элементов.

В табл. 8.21,6 показан список (8.10), распределенный по "карманам", в соответствии с правилом, когда число i вставляется в "карман" с номером $[i/n]$.

Таблица 8.5. **Двухэтапная „карманная" сортировка**

"Карман"	Содержимое	"Карман"	Содержимое
0	0	0	0, 1, 4, 9
1	1, 81	1	16
2		2	25
3		3	36
4	64, 4	4	49
5	25	5	
6	36, 16	6	64
7		7	
8		8	81
9	9, 49	9	
	а		б

Чтобы увидеть, почему этот алгоритм работает правильно, достаточно заметить, что когда числа помещаются в один "карман", например числа О, 1, 4 и 9 — в карман 0, то они будут располагаться в возрастающем порядке, поскольку в списке (8.10) они упорядочены по самой правой цифре. Следовательно, в любом "кармане" числа также будут упорядочены по самой правой цифре. И, конечно, распределение чисел на втором этапе по "карманам" в соответствии с первой цифрой гарантирует, что в конечном объединенном списке все числа будут расставлены в возрастающем порядке.

Покажем правильность работы описанного алгоритма в общем случае, предполагая, что сортируемые числа являются произвольными двузначными целыми числами из интервала от 0 до $n^2 - 1$. Рассмотрим целые числа $i = an + b$ и $j = cn + d$, где все числа a, b, c и d из интервала от 1 до $n - 1$. Пусть $i < j$. Тогда неравенство $a > c$ невозможно, поэтому $a \le c$. Если $a < c$, тогда на втором этапе сортировки число i будет находиться в "кармане" с меньшим номером, чем номер "кармана", в котором находится число j, и, следовательно, в конечном списке число i будет предшествовать числу j. Если $a = c$, то число b должно быть меньше, чем d. Тогда в объединенном списке после первого этапа сортировки число i будет предшествовать числу j, поскольку число i находилось в "кармане" с номером b, а число j — в кармане с номе-

ром d. На втором этапе числа i и j будут находиться в одном "кармане" (так как $a = c$), но число i будет вставлено в список этого "кармана" раньше, чем число j. Поэтому и в конечном списке число i будет предшествовать числу j. □

Общая поразрядная сортировка

Предположим, что тип данных ключей keytype является записями со следующими полями:

```
type
    keytype = record
        day: 1..31;
        month: (jan, ..., dec);
        year: 1900..1999
    end;                                        (8.11)
```

либо массивом элементов какого-нибудь типа, например

```
type
    keytype = array[1..10] of char;             (8.12)
```

Далее будем предполагать, что данные типа keytype состоят из k компонент f_1, fa, ..., f_k типа t_1, t_2 . . . t_k. Например, в (8.11) t_1 = **1..31**, t_2 = (**jan**, ..., dec), а t_3 = **1900..1999**. В (8.12) $k = 10$, а $t_1 = t_2 = \cdots = t_k$ = char.

Предположим также, что мы хотим сортировать записи в *лексикографическом порядке* их ключей. В соответствии с этим порядком ключевое значение $(a_1, a_2, ..., a_k)$ меньше ключевого значения $(b_1, b_2, ..., b_k)$, где a_i и fc, — значения из поля f_i (i = 1, 2, ..., k), если выполняется одно из следующих условий:

1. $a_1 < b_1$, **или**
2. $a_1 = b_1$ и $a_2 < b_2$, **или**

   .
   .
   .

k. $a_1 = b_1$, $a_2 - b_2$, ..., a_{k-1}= fet_j и $a_k < b_k$.

Другими словами, ключевое значение $(a_1, a_2, ..., a_k)$ меньше ключевого значения $(b_1, b_2, ..., b_k)$, если существует такой номер j ($1 < j \leq k$ - 1), что $a_1 = b_1$, a_2 = 62» ···› a_j=fy и $a_{j+1} < b_{j+1}$.

Если принять сделанные выше определения ключей, то в этом случае значения ключей можно рассматривать как целочисленные выражения в некоторой системе счисления. Например, определение (8.12), где каждое поле ключа содержит какой-либо символ, позволяет считать эти символы целочисленными выражениями по основанию 128 (если использовать только латинские буквы) или, в зависимости от используемого набора символов, по какому-то другому основанию. В определении типов (8.11) значения поля *year* (год) можно считать целыми числами по основанию 100 (так как здесь значения поля изменяются от 1900 до 1999, т.е. могут изменяться только последние две цифры), значения поля *month* (месяц) очевидно можно интерпретировать как целочисленные выражения по основанию 12, а значения третьего поля *day* (день) — как целочисленные выражения по основанию 31. Предельный случай такого подхода: любые целые числа (из конечного фиксированного множества) рассматриваются как массивы цифр по основанию 2 или другому подходящему основанию. Обобщение "карманной" сортировки, использующее такое видение значений ключевых полей, называется *поразрядной сортировкой.*

Основная идея поразрядной сортировки заключается в следующем: сначала проводится "карманная" сортировка всех записей по полю f_k (как бы по "наименьшей значащей цифре"), затем отсортированные по "карманам" записи объединяются (наименьшие значения идут первыми), затем к полученному объединенному списку

применяется "карманная" сортировка по полю f_{k-1}, снова проводится конкатенация содержимого "карманов", к объединенному списку применяется "карманная" сортировка по полю f_{k-2}, и т.д. Так же, как и в примере 8.6, при сортировке записей по "карманам" новая запись, вставляемая в карман, присоединяется в конец списка элементов этого "кармана", а не в начало, как было в "чистой карманной" сортировке. После выполнения "карманной" сортировки по всем ключам $f_k, f_{k-1}, ..., f_1$ и последнего объединения записей получим их результирующий список, где все записи будут упорядочены в лексикографическом порядке в соответствии с этими ключами. Эскиз описанного алгоритма в виде процедуры ***radixsort*** (поразрядная сортировка) приведен в листинге 8.13. Для обоснования этого алгоритма можно применить методику, показанную в примере 8.6.

Листинг 8.13. Поразрядная сортировка

```
    procedure radixsort;
        { radixsort сортирует список A из п записей по ключевым полям
          f\, f2, ..., fk типов fci, t2, ..., tk соответственно. В качестве
          "карманов" используются массивы Bi типа
          array[ti] of listtype, 1 < i < k, где  listtype — тип данных
          связанных списков записей }
      begin
(1)       for i:= k downto 1 do begin
(2)          for для каждого значения v типа ti do
                 { очистка "карманов" }
(3)              сделать Bi[v] пустым;
(4)          for для каждой записи r из списка A do
(5)              переместить запись r в конец списка "кармана" Bi[v],
                   где v — значение ключевого поля fi записи r;
(6)          for для каждого значения v типа tj
                                          в порядке возрастания v do
(7)              конкатенация Bi[v] в конец списка A
          end
      end; { radixsort }
```

Анализ поразрядной сортировки

Предполагаем, что для эффективного выполнения поразрядной сортировки применяются подходящие структуры данных. В частности, предполагаем, что список сортируемых элементов представлен в виде связанного списка, а не массива. Но на практике можно использовать и массив, если добавить еще поле связи типа recordtype для связывания элементов $A[i]$ и $A[i + 1]$ для $i = 1, 2, ..., n - 1$. Таким способом можно создать связанный список в массиве A за время порядка $O(n)$. Отметим также, что при таком представлении элементов нет необходимости в их копировании в процессе выполнения алгоритма, так как для перемещения записей из одного списка в другой достаточно изменить значения поля связи.

Как и ранее, для быстрого выполнения операции конкатенации будем использовать указатели на конец каждого списка. Тогда цикл в строках (2), (3) листинга 8.13 выполняется за время $O(s_i)$, где s_i — число различных значений типа t_i. Цикл строк (4), *(5)* требует времени порядка $O(n)$, цикл строк (6), (7) — $O(s_i)$. Таким образом, общее время выполнения поразрядной сортировки составит $\sum_{i=1}^{k} O(s_i + n)$, т.е. $O(kn + \sum_{i=1}^{k} s_i)$, или, если k константа, $O(n + \sum_{i=1}^{k} s_i)$.

Пример 8.7. Если ключи являются целыми числами из интервала от 0 до л* - 1 для некоторой константы k, то можно обобщить пример 8.6 и рассматривать ключи как целые числа по основанию n, состоящие из не более чем k "цифр". Тогда для

всех i, $1 < i < k$, t_i — целые числа из интервала от О до $n - 1$ и $s_i = n$. В этом случае выражение $O(n + \sum_{i=1}^{k} s_i)$ примет вид $O(n + kn)$, которое имеет порядок $O(n)$, поскольку k — константа.

Другой пример: если ключи являются символьными строками фиксированной длины k, тогда для всех i $s_i = 128$ (например) и $\sum_{i=1}^{k} s_i$ также будет константой. Таким образом, алгоритм поразрядной сортировки по символьным строкам фиксированной длины выполняется за время $O(n)$. Фактически, если k — константа и все s_i являются константами (или если даже имеют порядок роста $O(n)$), то в этом случае поразрядная сортировка выполняется за время порядка $O(n)$. Но если k растет вместе с ростом n, то время выполнения поразрядной сортировки может отличаться от $O(n)$. Например, если ключи являются двоичными числами длины log га, тогда А = log n и $s_i = 2$ для всех i. В таком случае время выполнения алгоритма составит $O(n \log n)$.[1] П

8.6. Время выполнения сортировок сравнениями

Существует "общеизвестная теорема" (из математического фольклора), что для сортировки n элементов "требуется времени $n \log n$". Как мы видели в предыдущем разделе, это утверждение не всегда верно: если тип ключей такой, что значения ключей выбираются из конечного (по количеству элементов) множества (это позволяет с "пользой" применить метод "карманной" или поразрядной сортировки), тогда для сортировки достаточно времени порядка $O(n)$. Однако в общем случае мы не можем утверждать, что ключи принимают значения из конечного множества. Поэтому при проведении сортировки мы можем опираться только на **операцию** сравнения двух значений ключей, в результате которой можно сказать, что одно ключевое значение меньше другого.

Читатель может заметить, что во всех алгоритмах сортировки, которые мы рассмотрели до раздела 8.5, истинный порядок элементов определялся с помощью последовательных сравнений значений двух ключей, затем, в зависимости от результата сравнения, выполнение алгоритма шло по одному из двух возможных путей. В противоположность этому, алгоритм, подобный показанному в примере 8.5, за один шаг распределяет n элементов с целочисленными значениями ключей по n "карманам", причем номер "кармана" определяется значением ключа. Все программы раздела 8.5 используют мощное средство (по сравнению с простым сравнением значений) языков программирования, позволяющее за один шаг находить в массиве по индексу местоположение нужной ячейки. Но это мощное средство невозможно применить, если тип данных ключей соответствует, например, действительным числам. В языке Pascal и в большинстве других языков программирования нельзя объявить индексы массива как действительные числа. И если даже мы сможем это сделать, то операцию **конкатенации** "карманов", номера которых являются машинно-представимыми действительными числами, невозможно будет выполнить за приемлемое время.

Деревья решений

Сосредоточим свое внимание на алгоритмах сортировки, которые в процессе сортировки используют только сравнения значений двух ключей. Можно нарисовать двоичное дерево, в котором узлы будут соответствовать "состоянию" программы после определенного количества сделанных сравнений ключей. Узлы дерева можно

[1] Но в этом случае, если двоичные числа длины $\log n$ можно трактовать как одно слово, лучше объединить поля ключей в одно поле типа **1..n** и использовать обычную "карманную" сортировку.

также рассматривать как соответствие некоторым начальным упорядочиваниям данных, которые "привели" программу сортировки к такому "состоянию". Можно также сказать, что данное программное "состояние" соответствует нашим знаниям о первоначальном упорядочивании исходных данных, которые можно получить на данном этапе выполнения программы.

Если какой-либо узел соответствует нескольким упорядочиваниям исходных данных, то программа на этом этапе не может точно определить истинный порядок данных и поэтому необходимо выполнить следующее сравнение ключей, например "является ли k_1 < fe_2?". В зависимости от возможных ответов на этот вопрос создаются два сына этого узла. Левый сын соответствует случаю, когда значения ключей удовлетворяют неравенству ft_t < k_2; правый сын соответствует упорядочению при выполнении **неравенства** $k_1 > k_2$.[1] Таким образом, каждый сын "содержит" информацию, доступную родителю, плюс сведения, вытекающие из ответа на вопрос, "является ли $k_1 < k_2$?".

Пример 8.8. Рассмотрим алгоритм сортировки вставками при $n = 3$. Предположим, что элементы *A*[1], *A*[2] и *A*[3] имеют значения ключей *a, Ь* и *c* соответственно. Три **элемента** а, *b* и *c* можно упорядочить шестью различными способами, поэтому дерево решений начнем строить с узла, представляющего все возможные упорядочивания и помеченного на рис. 8.5 цифрой (1). Алгоритм сортировки вставками сначала сравнивает значения ключей элементов *A*[1] и А[2], т.е. сравнивает значения *a* и *b*.[2] Если *Ь* меньше *a,* то возможны только три правильных упорядочивания *Ьас, Ьса* и *cba*, где *Ь* предшествует *a*. Эти три упорядочивания соответствуют узлу (2) на рис. 8.5. Другие три упорядочивания (если *a* меньше *Ь,* т.е. *a* предшествует *Ь)* соответствуют узлу (3), правому сыну узла (1).

Теперь посмотрим, что делать дальше, если исходные данные таковы, что в процессе сортировки мы достигли узла (2). Алгоритм сортировки вставками поменяет местами элементы *A*[1] и *A*[2], после чего получим текущее упорядочение *bac*. Далее проводится сравнениэ элементов *A*[3] и *A*[2]. Так как теперь А[2] содержит элемент с ключом *a,* то сравниваются ключи а и с. Узел (4) соответствует двум упорядочиваниям из узла (2), когда значение *c* предшествует значению а, узел (5) соответствует случаю, когда *a* меньше *c*.

Если алгоритм достиг узла (5), то сортировка заканчивается, поскольку этому узлу соответствует одно упорядочивание элементов *bac*. Если алгоритм находится в "состоянии" узла (4), т.е. неравенство *A[3]* < *A*[2] истинно, тогда эти элементы переставляются местами: получаем текущее упорядочение *bca*. Далее сравниваются А[1] и *A*[2], т.е. значения ключей *c* и *Ь*. Узлы (8), (9) соответствуют двум возможным вариантам отношения между значениями *c* и *Ъ*. Если состояние программы соответствует узлу (8), то опять переставляются элементы *A*[1] и *A*[2]. В каком бы из узлов ((8) или (9)) ни находилась программа сортировки, ее выполнение заканчивается, так как этим узлам (точнее, листьям дерева решений) соответствуют по одному упорядочению элементов.

Поддерево, исходящее из узла (3), полностью симметрично рассмотренному нами поддереву, исходящему из узла (2), поэтому его описание мы опускаем. Дерево решений, показанное на рис. 8.5, позволяет найти правильный порядок расположения записей, соответствующий значениям ключей *a, b я c,* поскольку все его листья соответствуют какому-либо одному упорядочению. П

---

[1] Можно предполагать, что значения всех ключей различны, поскольку если мы можем сортировать элементы с различными ключами, то, несомненно, сможем упорядочить элементы и с совпадающими ключами.

[2] Здесь а, Ь и с не "буквы", а буквенное обозначение значений ключей. Поэтому может быть *a* < *Ь* и *Ь* < *a*. — *Прим. ред.*

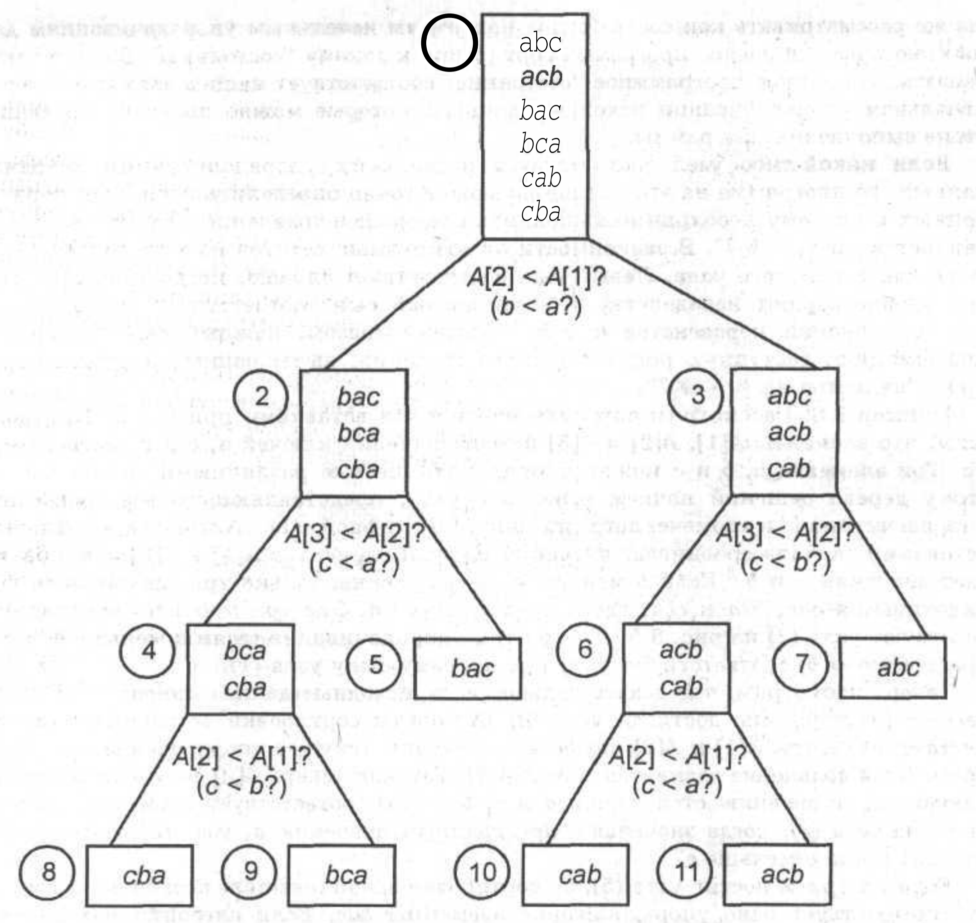


Рис. 8.5. Дерево решений для алгоритма сортировки вставками

Размер дерева решений

Дерево на рис. 8.5 имеет шесть листьев, соответствующих шести возможным упорядочениям исходного списка из трех элементов а, & и с. В общем случае, если сортируется список из *n* элементов, тогда существует п! = $1 \cdot 2 \cdot \ldots \cdot (n-2) \cdot (n-1) \cdot n$ возможных *исходов,* которые соответствуют различным упорядочениям исходного списка элементов $a_1, a_2, \ldots, a_n$. В самом деле, на первую позицию можно поставить любой из *n* элементов, на вторую — любой из оставшихся *n* - 1 элементов, на третью — любой из *n* - *2* элементов и т.д., произведение количества всех этих частных исходов дает $n!$. Таким образом, любое дерево решений, описывающее алгоритм сортировки списка из *n* элементов, должно иметь не менее га! листьев. Фактически, если удалить узлы и листья, соответствующие невозможным сочетаниям элементов, получим в точности л! листьев.

Двоичные деревья, имеющие много листьев, должны иметь и длинные пути. Длина пути от корня к листу равна количеству сравнений, которые необходимо сделать, чтобы получить то упорядочение, которое соответствует данному листу, исходя из определенного начального списка элементов *L*. Поэтому длина самого длинного пути от корня к листу — это нижняя граница количества шагов (количество сравнений ключей), вы-

полняемых алгоритмом сортировки в самом худшем случае. Но также не надо забывать, что, кроме сравнения ключей, алгоритм выполняет и другие операции.

Рассмотрим, какова же может быть длина путей в двоичном дереве с k листьями. Двоичное дерево, в котором все пути имеют длину p или меньше, может иметь 1 корень, 2 узла на первом уровне, 4 узла на втором уровне и далее 2^i узлов на уровне i. Поэтому наибольшее число листьев 2^p на уровне p, если нет узлов, выше этого уровня. Отсюда следует, что двоичное дерево с k листьями должно иметь путь длиной не менее $\log k$. Если положить $k = n!$, то получим, что любой алгоритм сортировки, использующий для упорядочивания n элементов только сравнения значений ключей, в самом худшем случае должен выполняться за время, не меньшее $\Omega(\log(n!))$.

Какова степень роста величины $\log(n!)$? По формуле Стирлинга $n!$ можно достаточно точно аппроксимировать функцией $(n/e)^n$, где e — 2.7183... — основание натурального логарифма. Поскольку $\log((n/e)^n) = n \log n - n \log e = n \log n$ - 1.44л, то отсюда получаем, что $\log(n!)$ имеет порядок $n \log n$. Можно получить более простую нижнюю границу, если заметить, что л! является произведением не менее и/2 сомножителей, каждое из которых не превышает л/2. Поэтому га! > $(n/2)^{n/2}$. Следовательно, $\log(n!) > (n/2)\log(n/2) = (n/2)\log n$ - л/2. Из любой приведенной оценки вытекает, что сортировка посредством сравнений требует в самом худшем случае времени порядка $\Omega(n \log n)$.

Анализ времени выполнения в среднем

Можно ожидать, что алгоритм, который в самом худшем случае выполняется за время $O(n \log n)$, в среднем будет иметь время выполнения порядка $O(n)$ или, по крайней мере, меньшее, чем $O(n \log n)$. Но это не так, что мы сейчас и покажем, оставляя детали доказательства читателю в качестве упражнения.

Надо доказать, что в произвольном двоичном дереве с k листьями средняя глубина листьев не менее $\log k$. Предположим, что это не так, и попробуем построить контрпример двоичного дерева T с k листьями, у которого средняя глубина листьев была бы меньше $\log k$. Поскольку дерево T не может состоять только из одного узла (в этом случае невозможен контрпример, так как утверждение выполняется — средняя глубина равна 0), пусть $k \geq 2$. Возможны две формы двоичного дерева T, которые показаны на рис. 8.6.

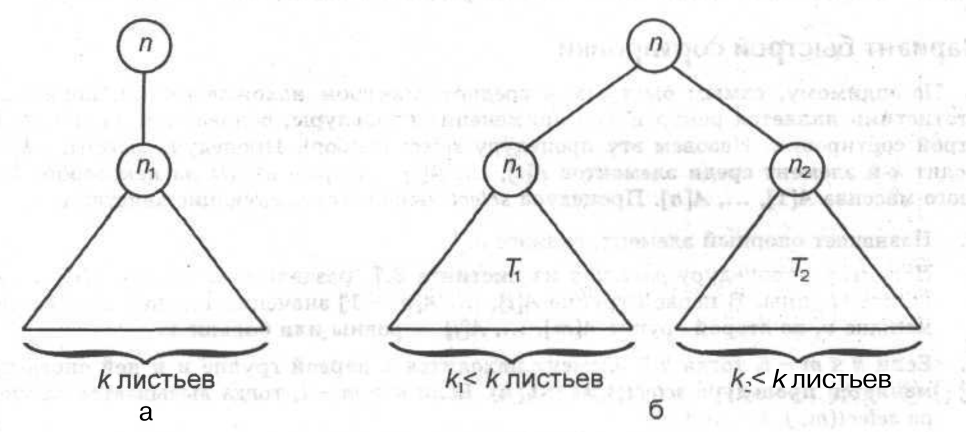


Рис. *8.6. Возможные формы двоичного дерева T*

Дерево на рис. **8.6,а** не может быть контрпримером с минимальной средней глубиной листьев, поскольку дерево с корнем в узле n_1 имеет столько же листьев, что и дерево T, но его средняя глубина меньше. Дерево на рис. **8.6,б** не нарушает

требования к контрпримеру. Пусть средняя глубина листьев поддерева T_1 составляет $\log k_1$, а средняя глубина листьев поддерева T_2 — $\log k_2$. Тогда средняя глубина листьев дерева T равна

$$\left(\frac{k_1}{k_1 + k_2}\right)\log(k_1) + \left(\frac{k_2}{k_1 + k_2}\right)\log(k_2) + 1.$$

Поскольку $k_1 + k_2 = k$, то последнее выражение можно переписать так:

$$\frac{1}{k}(k_1 \log(2k_1) + k_2 \log(2k_2)). \tag{8.13}$$

Легко проверить, что при $k_1 = k_2 =$ ft/2 выражение (8.13) принимает значение $\log k$. Теперь читатель должен показать, что выражение (8.13) при выполнении условия $k_1 + k_2 = k$ имеет минимум, когда fti $= k_2$. Это несложное упражнение для читателя, знакомого с дифференциальным исчислением. Так как минимум выражения (8.13) равен $\log k$, то можно считать доказанным, что контрпример дерева T со средней глубиной листьев, меньшей чем $\log k$, не существует.

8.7. Порядковые статистики

Задача вычисления *порядковых статистик* заключается в следующем: дан список из n записей и целое число k, необходимо найти ключ записи, которая стоит в списке, отсортированном в возрастающем порядке, в fe-й позиции. Для краткости эту задачу будем называть "задачей нахождения ft-й порядковой статистики". Специальные случаи этой задачи возникают при $k = 1$ (нахождение минимального элемента), $k = n$ (нахождение максимального элемента) и, если n нечетно, $k = (n + 1)/2$ (нахождение *медианы).*

В некоторых случаях решение задачи вычисления порядковых статистик находится за линейное время. Например, нахождение минимального элемента (как и максимального) требует времени порядка $O(n)$. Как упоминалось при изучении пирамидальной сортировки, если $k < n/\log n$, тогда для нахождения ft-й порядковой статистики можно построить кучу (за время порядка $O(n)$) и затем выбрать из нее k наименьших элементов за время $O(n + k \log n) = O(n)$. Подобным образом при $k > n - n/\log n$ также можно найти k-ю порядковую статистику за время $O(n)$.

Вариант быстрой сортировки

По-видимому, самым быстрым в среднем способом нахождения ft-й порядковой статистики является рекурсивное применение процедуры, основанной на методе быстрой сортировки. Назовем эту процедуру *select* (выбор). Процедура *select(i. j, k)* находит ft-й элемент среди элементов $A[i]$, ..., $A[j]$, которые взяты из некоторого большого массива $A[1]$, ..., $A[n]$. Процедура *select* выполняет следующие действия.

1. Назначает опорный элемент, скажем v.
2. Используя процедуру *partition* из листинга 8.7, разделяет элементы $A[i]$, ..., $A[j]$ на две группы. В первой группе $A[i]$, ..., $A[m - 1]$ значения ключей всех записей меньше v, во второй группе $A[m]$, ..., $A[j]$ — равны или больше v.
3. Если $k < m - i$, тогда k-й элемент находится в первой группе и к ней снова применяется процедура *select(i, m* - 1, *k).* Если $k > m - i$, тогда вызывается процедура *select(m,j, k - m + i).*

Рано или поздно вызов процедуры *select(i, j, k)* будет сделан для элементов $A[i]$, ..., $A[j]$, имеющих одинаковые значения ключей (и поэтому скорее всего будет $i = j$). Значение ключа этих элементов и будет искомым значением fe-й порядковой статистики.

Так же, как и метод быстрой сортировки, процедура *select* (так как она описана выше) в самом худшем случае потребует времени не менее $\Omega(n^2)$. Например, если при поиске минимального элемента в качестве опорного элемента всегда брать наибольшее из возможных значений ключей, то получим для этой процедуры время выполнения порядка $O(n^2)$. Но в среднем процедура *select* работает значительно быстрее, чем алгоритм быстрой сортировки, в среднем она выполняется за время $O(n)$. Принципиальная разница между этими алгоритмами заключается в том, что когда процедура быстрой сортировки вызывается два раза, процедура *select* в аналогичной ситуации вызывается только один раз. Можно провести анализ процедуры *select* теми же методами, которые применялись при анализе алгоритма быстрой сортировки, но чтобы избежать сложных математических выкладок, здесь мы ограничимся только интуитивными соображениями, показывающими среднее время выполнения процедуры *select*. Эта процедура повторно вызывает себя на подмножестве, которое является только частью того множества, для которого она вызывалась на предыдущем шаге. Сделаем умеренное предположение, что каждый вызов этой процедуры осуществляется на множестве элементов, которое составляет 9/10 того множества, для которого был произведен предыдущий вызов процедуры. Тогда, если обозначить через $T(n)$ время, затрачиваемое процедурой *select* на множестве из n элементов, для некоторой константы c будем иметь

$$T(n) \le T\left(\frac{9}{10}n\right) + cn. \qquad (8.14)$$

Используя технику из следующей главы, можно показать, что решением неравенства (8.14) будет $T(n) = O(n)$.

Линейный метод нахождения порядковых статистик

Чтобы гарантировать для процедуры, подобной *select,* в самом худшем случае время выполнения порядка $O(n)$, необходимо показать, что за линейное время можно выбрать такой опорный элемент, который разбивает множество элементов на два подмножества, размер которых не больше некоторой фиксированной доли исходного множества. Например, решение неравенства (8.14) показывает, что если опорный элемент не меньше $(n/10)$-го порядкового элемента и не больше $(9n/10)$-го порядкового элемента, тогда множество, для которого вызывается процедура *select,* разбивается на подмножества, не превышающие 9/10 исходного множества, и это гарантирует линейное время в самом худшем случае.

Нахождение "хорошего" опорного элемента можно осуществить посредством следующих двух шагов.

1. Выполняется разделение n элементов на группы по 5 элементов, оставляя в стороне группу из 1 - 4 элементов, не вошедших в другие группы. Каждая группа из 5 элементов сортируется любым алгоритмом в порядке возрастания и берутся средние элементы из каждой группы. Всего таких средних элементов будет [л/5].
2. Используя процедуру *select,* находится медиана этих средних элементов. Если [л/5] — четно, то берется элемент, наиболее близкий к середине. В любом случае будет найден элемент, стоящий в позиции $[(n + 5)/10]$ отсортированного списка средних элементов.

Если среди записей не слишком много таких, значения ключей которых совпадают со значением опорного элемента, тогда значение опорного элемента достаточно далеко от крайних значений ключей. Для простоты временно положим, что все значения ключей различны (случай одинаковых ключевых значений будет рассмотрен ниже). Покажем, что выбранный опорный элемент (являющийся $[(n + 5)/10]$-м порядковым элементом среди [га/5] средних элементов) больше не менее $3[(n - 5)/10]$ элементов из га исходных элементов. В самом деле, опорный элемент больше $[(n - 5)/10]$ средних элементов. В свою очередь каждый из этих средних элементов больше двух элементов из той

пятерки элементов, которой он принадлежит. Отсюда получаем величину $3[(n - 5)/10]$. Если га > 75, тогда $3[(n - 5)/10]$ не меньше ге/4. Подобным образом доказывается, что выбранный опорный элемент меньше или равен не менее $3[(n - 5)/10]$ элементов. Отсюда следует, что при $n > 75$ выбранный опорный элемент находится между первой и последней четвертями отсортированного списка исходных элементов.

Алгоритм нахождения **k-й** порядковой статистики в виде функции *select* показан в листинге 8.14. Как и в алгоритмах сортировки, предполагаем, что список исходных элементов представлен в виде массива A записей типа recordtype и что записи имеют поле *key* типа **key type.** Нахождение **k-й** порядковой статистики выполняется, естественно, посредством вызова функции ***select*(1, *n*, *k*)**.

Листинг 8.14. Линейный алгоритм нахождения *k*-й порядковой статистики

```
        function select (i, j, k: integer ): keytype;
            { Функция возвращает значение ключа k-го элемента
                из A[i], ..., A[j] }
            var
                m: integer; { используется в качестве индекса }
            begin
(1)             if j - i < 74 then begin
                    { элементов мало, select рекурсивно не применяется }
(2)                 сортировка A[i], ..., A[j] простым алгоритмом;
(3)                 return(A[i + k - 1].key)
                end
                else begin { select применяется рекурсивно }
(4)                 for m:= 0 to (j - i - 4) div 5 do
                        { нахождение средних элементов в группах
                          из 5-ти элементов }
(5)                     нахождение 3-го по величине элемента в группе
                          A[i + 5*m], ..., A[i + 5*m + 4] и
                          перестановка его с A[i + m];
(6)                 pivot:= select(i,i+(j-i-4) div 5,(j-i-4) div 10);
                        { нахождение медианы средних элементов }
(7)                 m:= partition(i, j, pivot);
(8)                 if k <= m - i then
(9)                     return(select(i, m - 1, k))
                    else
(10)                    return(select(m, j, k - (m - i)))
                end
            end; { select }
```

Для анализа времени выполнения процедуры *select* из листинга **8.14** положим здесь, что $j - i + 1$ = га. Строки (2), (3) выполняются, если n не превосходит 74. Поэтому, если даже выполнение строки (2) требует времени порядка $O(n^2)$ в общем случае, здесь потребуется конечное время, не превосходящее некоторой константы c. Таким образом, при $n < 74$ строки (1) — (3) требуют времени, не превосходящего некоторой константы c_1.

Теперь рассмотрим строки (4) - (10). Строка (7), разбиение множества элементов по опорному элементу, имеет время выполнения $O(n)$, как было показано при анализе алгоритма быстрой сортировки. Тело цикла (4), (5) выполняется $n/5$ раз, упорядочивая за каждую итерацию 5 элементов, на что требуется фиксированное время. Поэтому данный цикл выполняется за время порядка $O(n)$.

Обозначим через $T(n)$ время выполнения процедуры *select* на n элементах. Тогда строка (6) требует времени, не превышающего $T(n/5)$. Поскольку строку (10) можно достигнуть только для га > 75, а, как было показано ранее, при $n > 75$ количество элементов, меньших опорного элемента, не превышает Зга/4. Аналогично, количество

элементов, больших или равных опорному элементу, не превышает $3n/4$. Следовательно, время выполнения строки (10) не превышает $T(3n/4)$. Отсюда вытекает, что для некоторых констант c_1 и c_2 справедливы неравенства

$$T(n) \le \begin{cases} c_1, & \text{если га} < 74, \\ c_2n + T(n/5) + T(3n/4), & \text{если } n > 75. \end{cases} \quad (8.15)$$

В неравенствах (8.15) выражение c_2n представляет время выполнения строк (1), (4), (5) и (7), выражение $T(n/5)$ соответствует времени выполнения строки (6), а $T(3n/4)$ — строк (9), (10).

Далее мы покажем, что $T(n)$ из (8.15) имеет порядок $O(n)$. Но сначала несколько слов о "магическом числе" 5, размере групп в строке (5) и условии $n < 75$, которое определяет способ нахождения порядковой статистики (посредством рекурсивного повторения функции *select или* прямым способом). Конечно, эти числа могут быть и другими, но в данном случае они выбраны так, чтобы в формуле (8.15) выполнялось неравенство $1/5 + 3/4 < 1$, необходимое для доказательства линейного порядка величины $T(n)$.

Неравенства (8.15) можно решить методом индукции по n. Предполагаем, что решение имеет вид cn для некоторой константы c. Если принять $c > c_1$, тогда $T(n) < cn$ для всех га от 1 до 74, поэтому рассмотрим случай, когда га > 75. Пусть, в соответствии с методом математической индукции, $T(m) < cm$ для всех m, $m < n$. Тогда из (8.15) следует, что

$$T(n) < c_2n + cn/5 + 3cn/4 < c_2n + 19cn/20. \quad (8.16)$$

Если положить $c — \max(c_1, 20c_2)$, тогда из (8.16) получаем $T(n) < cn/20 + cn/5 + 3cn/4 = cn$, что и требовалось доказать. Таким образом, $T(n)$ имеет порядок $O(n)$.

Случай равенства некоторых значений ключей

Напомним, что при создании процедуры листинга 8.14 мы предполагали, что все значения ключей различны. Это сделано из-за того, что в противном случае нельзя доказать, что множество элементов при разбиении распадется на подмножества, не превышающие Зга/4. Для возможного случая равенства значений ключей необходимо только немного изменить функцию *select* из листинга 8.14. После строки (7), выполняющей разбиение множества, надо добавить операторы, собирающие вместе все записи, имеющие значения ключей, совпадающих с опорным элементом. Пусть таких ключей будет $p > 1$. Если $m - j \le k \le m - i + p$, тогда в последующем рекурсивном вызове функции *select* нет необходимости — достаточно вернуть значение $A[m].key$. В противном случае строка (8) остается без изменений, а в строке (10) осуществляется вызов $select(m + p, j, k — (m - i) — p)$.

Упражнения

8.1. Есть восемь целых чисел 1, 7, 3, 2, 0, 5, 0, 8. Выполните их упорядочивание с помощью метода "пузырька", сортировки вставками, сортировки посредством выбора.

8.2. Есть шестнадцать целых чисел 22, 36, 6, 79, 26, 45, 75, 13, 31, 62, 27, 76, 33, 16, 62, 47. Трактуя каждое число как пару цифр из интервала 0-9, выполните их упорядочивание, используя быструю сортировку, сортировку вставками, пирамидальную сортировку и "карманную" сортировку.

8.3. Процедура *Shellsort* (сортировка Шелла[1]), показанная в листинге 8.15, сортирует элементы массива $A[1..n]$ следующим образом: на первом шаге упорядочиваются элементы га/2 пар $(A[i], A[n/2 + i])$ для $1 \le i \le n/2$, на втором

[1] Этот метод сортировки часто называют *сортировкой с убывающим шагом.*

шаге упорядочиваются элементы в л/4 группах из четырех элементов ($A[i], A[n/4 + i], A[n/2 + i], A[3n/4 + i]$) для $1 < i <$ л/4, на третьем **шаге** упорядочиваются элементы в л/8 группах из восьми элементов и т.д. На последнем шаге упорядочиваются элементы сразу во всем массиве *А.* На каждом шаге для упорядочивания элементов используется метод сортировки вставками.

Листинг 8.15. Сортировка Шелла

```
procedure Shellsort ( var A: array[1..n] of integer );
var
     i, j, incr: integer;
begin
     incr:= n div 2;
     while incr > 0 do begin
          for i:= incr + 1 to n do begin
               j:= i - incr;
               while j > 0 do
                    if A [ j ] > A ( j + incr] then begin
                         swap(A[j], A[j + incr]);
                         j:= j - incr
                    end
                    else
                         j:= 0 { останов }
          end;
          incr:= incr div 2
     end
end; { Shellsort }
```

а) с помощью сортировки Шелла выполните упорядочивание последовательностей целых чисел из упражнений 8.1 и 8.2;

*б) покажите, что если элементы $A[i]$ и $A[n/2^k + i]$) упорядочены (переставлены) на ***k*-м** шаге, то они останутся упорядоченными и на следующем ($k + 1$)-м шаге;

в) в листинге 8.15 использовалась убывающая последовательность шагов л/2, л/4, л/8, ..., 2, 1. Покажите, что алгоритм Шелла работает с любой убывающей последовательностью шагов, у которой последний шаг равен **1;**

****г)** покажите, что время выполнения алгоритма Шелла имеет порядок $O(n^{1.5})$.

***8.4.** Предположим, что необходимо сортировать список элементов, состоящий из уже упорядоченного списка, который следует за несколькими "случайными" элементами. Какой из рассмотренных в этой главе методов сортировки наиболее подходит для решения такой задачи?

***8.5.** Алгоритм сортировки называется *устойчивым,* если он сохраняет исходный порядок следования элементов с одинаковыми значениями ключей. Какой из алгоритмов сортировки, представленных в этой главе, устойчив?

***8.6.** Предположим, что в алгоритме быстрой сортировки в качестве опорного элемента выбирается первый элемент сортируемого подмножества.

а) какие изменения следует сделать в алгоритме быстрой сортировки (листинг 8.8), чтобы избежать "зацикливания" (бесконечного цикла) в случае последовательности равных элементов?

б) покажите, что измененный алгоритм быстрой сортировки в среднем будет иметь время выполнения порядка $O(n \log n)$.

8.7. Покажите, что любой алгоритм сортировки, перемещающий за один шаг только один элемент и только на одну позицию, должен иметь временную сложность не менее $\Omega(n^2)$.

8.8. В алгоритме пирамидальной сортировки процедура *pushdown* из листинга 8.10 строит частично упорядоченное дерево за время *O(n)*. Вместо того чтобы начинать построение дерева от листьев, начните построение от корня. Какова в этом случае будет временная сложность процедуры построения частично упорядоченного дерева?

*8.9. Допустим, что есть множество слов, т.е. строк, состоящих из букв *а* - *г*. Суммарная длина слов равна *n*. Покажите, что эти слова можно упорядочить за время *O(n)*. Отметим, что для слов одинаковой длины можно применить "карманную" сортировку. Но вы также должны предусмотреть случай слов разной длины.

*8.10. Покажите, что для сортировки вставками среднее время выполнения не меньше $\Omega(n^2)$.

**8.11. Рассмотрим алгоритм *случайной сортировки,* примененный к массиву $A[1..n]$ целых чисел: выбирается случайное число i из интервала от 1 до л и меняются местами элементы $A[1]$ и *A[i],* процесс повторяется до тех пор, пока не будут упорядочены все элементы массива. Каково ожидаемое время выполнения этой сортировки?

*8.12. В разделе 8.6 было показано, что сортировки, выполняемые посредством сравнений, в самом худшем случае требуют $\Omega(n \log n)$ сравнений. Покажите, что эта нижняя граница справедлива и в среднем.

*8.13. Докажите, что процедура *select,* неформально описанная в начале раздела 8.7, имеет среднее время выполнения *O(n)*.

8.14. Реализуйте оператор конкатенации CONCATENATE для структуры данных, показанной на рис. 8.4.

8.15. Напишите программу нахождения k наименьших элементов в массиве длиной *n*. Какова временная сложность этой программы? Для каких значений *k* эффективнее сначала выполнить сортировку всего массива, а затем взять *k* наименьших элементов, вместо поиска наименьших элементов в неупорядоченном массиве?

8.16. Напишите программу нахождения наибольшего и наименьшего элементов в массиве. Может ли эта программа обойтись менее чем 2га — 3 сравнениями?

8.17. Напишите программу нахождения *моды* (наиболее часто встречаемого элемента) в списке из *n* элементов. Какова временная сложность этой программы?

*8.18. Покажите на модели дерева решений из раздела 8.6, что любой алгоритм удаления дублирующих записей из заданного списка требует времени не менее $\Omega(n \log n)$.

*8.19. Предположим, что есть *k* множеств S_1, S_2 . . . S_k, каждое из которых состоит из га действительных чисел. Напишите программу получения списка всех сумм вида $s_1 + s_2 + ... + s_k$, где s_i принадлежит отсортированному множеству S_i. Какова временная сложность этой программы?

8.20. Предположим, есть сортированный массив строк s_1, s_2, ..., s_n. Напишите программу, определяющую, является ли предъявленная строка x членом этой последовательности. Какова временная сложность этой программы как функции от п и длины строки x?

Библиографические примечания

Монография [65] является исчерпывающим обзором по методам сортировки. Метод быстрой сортировки предложен Хоаром (Hoare) [49], более поздние варианты этой сортировки приведены в [37] и [103]. Пирамидальная сортировка была открыта Вильямсом (Williams) [119] и обоснована Флойдом (Floyd) [34]. Сложность дерева решений для сортировки изучалась в работе [36]. Линейный алгоритм сортировки из раздела 8.7 взят из статьи [13].

Алгоритм Шелла описан в [101], анализ его выполнения приведен в работе [87]. См. также книгу [3], где приведено одно решение упражнения 8.9.

ГЛАВА 9

Методы анализа алгоритмов

Что собой представляет "хороший" алгоритм? Не существует простого ответа на этот вопрос. Многие оценки алгоритмов включают такие субъективные понятия, как простота, понятность или соответствие ожидаемым входным данным. Более объективной (но, возможно, не самой важной) оценкой алгоритма может служить временная эффективность (время выполнение) алгоритма. В разделе 1.5 описывались основные методы определения времени выполнения простых программ. Но в более сложных **ситуациях,** например при оценке рекурсивных программ, необходимы другие методы. В этой короткой главе рассмотрены некоторые достаточно общие методы решения рекуррентных соотношений, которые возникают при анализе времени выполнения рекурсивных алгоритмов.

9.1. Эффективность алгоритмов

Один из способов определения временной эффективности алгоритма заключается в следующем: на основе данного алгоритма надо написать программу и измерить время ее выполнения на определенном компьютере для выбранного множества входных данных. Хотя такой способ популярен и, безусловно, полезен, он порождает определенные проблемы. Определяемое здесь время выполнения зависит не только от используемого алгоритма, но также от архитектуры и набора внутренних инструкций данного компьютера, от качества компилятора, да и от уровня программиста, реализовавшего тестируемый алгоритм. Время выполнения также может существенно зависеть от выбранного множества тестовых входных данных. Эта зависимость становится совершенно очевидной при реализации одного и того же алгоритма с использованием различных компьютеров, различных компиляторов, при привлечении программистов разного уровня и при использовании различных тестовых входных данных. Чтобы повысить объективность оценок алгоритмов, ученые, работающие в области компьютерных наук, приняли асимптотическую **временную** сложность как основную меру эффективности выполнения алгоритма. Термин *эффективность* является синонимом этой меры и относится в основном к времени выполнения в самом худшем случае (в отличие от среднего времени выполнения).

Напомним читателю определения $O(f(n))$ и $\Omega(f(n))$,данные в главе 1. Говорят, что алгоритм имеет эффективность (т.е. **временную** сложность в самом худшем случае) $O(f(n))$,или просто $f(n)$, если функция от n, равная максимуму числу шагов, выполняемых алгоритмом, имеет порядок роста $O(f(n))$, причем максимум берется по всем входным данным длины п. Можно сказать по-другому: существует константа c, такая, что для достаточно больших n величина $cf(n)$является верхней границей количества шагов, выполняемых алгоритмом для любых входных данных длины n.

Существует еще "подтекст" в утверждении, что "эффективность данного алгоритма есть $f(n)$": время выполнения алгоритма также равно $\Omega(f(n))$, т.е. $f(n)$ является функцией от га с наименьшей степенью роста, которая ограничивает сверху время выполнения алгоритма в самом худшем случае. Но последнее условие (о наименьшей степени роста) не является частью определения $O(f(n))$, часто вообще невозможно сделать никакого заключения о наименьшей степени роста верхней границы.

Наше определение эффективности алгоритма игнорирует константы пропорциональности, которые участвуют в определениях $O(f(n))$и $\Omega(f(n))$, и для этого есть весомые причины. Во-первых, поскольку большинство алгоритмов записываются на языках программирования высокого уровня, нам приходится описывать их в терми-

нах "шагов", каждый из которых должен выполняться за конечное фиксированное время после трансляции их в машинный язык какого-нибудь компьютера. Однако, как можно точно определить время выполнения какого-либо шага алгоритма, когда оно зависит не только от "природы" самого этого шага, но также от процесса трансляции и машинных инструкций, заложенных в компьютере? Поэтому вместо точной оценки эффективности алгоритма, пригодной только для конкретной машины (и для получения которой надо ещё пробраться через запутанные подробности машинных вычислений), используют менее точные (с точностью до константы пропорциональности), но более общие оценки эффективности в терминах $O(f(n))$.

Вторая причина, по которой используются асимптотические оценки и игнорируются константы пропорциональности, заключается в том, что асимптотическая временная сложность алгоритма в большей степени, чем эти константы, определяет граничный размер входных данных, которые можно обрабатывать посредством данного алгоритма. В главе 1 эта ситуация исследована подробно. С другой стороны, читатель должен быть готовым к тому, что при анализе некоторых важных задач (например, сортировки) мы, возможно, будем применять такие утверждения, как "алгоритм *A* на типовом компьютере выполняется в два раза быстрее, чем алгоритм *B*".

Возможны ситуации, когда мы будем отходить от времени выполнения в самом худшем случае как меры эффективности алгоритма. В частности, если известно ожидаемое распределение входных данных, на которых выполняется алгоритм, то в этой ситуации среднестатистическое время выполнения алгоритма является более разумной мерой эффективности по сравнению с оценкой времени выполнения в самом худшем случае. Например, в предыдущей главе мы анализировали среднее время выполнения алгоритма быстрой сортировки в предположении, что все первоначальные упорядочения входных последовательностей равновероятны.

9.2. Анализ рекурсивных программ

В главе 1 мы показали методы анализа времени выполнения программ, не использующих рекурсивных вызовов. Анализ рекурсивных программ значительно сложнее и, как правило, требует решения дифференциальных уравнений. Мы будем использовать другие методы, которые похожи на методы решения дифференциальных уравнений, и даже позаимствуем их терминологию.

Сначала рассмотрим пример процедуры сортировки (листинг 9.1). Эта процедура-функция ***mergesort*** (сортировка слиянием) в качестве входных данных использует список элементов длиной *n* и возвращает этот список отсортированным (выходные данные). Эта функция использует также процедуру *merge* (слияние), у которой входными данными являются два отсортированных списка L_1 и L_2. Процедура *merge*(L_1, L_2) просматривает эти списки поэлементно, начиная с наибольших элементов. На каждом шаге наибольший элемент из двух сравниваемых (наибольшие элементы из списков L_1 и L_2) удаляется из своего списка и помещается в выходные данные. В результате получается один отсортированный список, содержащий все элементы из списков L_1 и L_2. Детали выполнения процедуры *merge* сейчас для нас не имеют значения, сортировка слиянием подробно рассмотрена в главе 11. Сейчас для нас важно только то, что процедура *merge* на списках длиной $n/2$ выполняется за время порядка $O(n)$.

Листинг 9.1. Рекурсивная процедура сортировки слиянием

```
function mergesort ( L: LIST; n: integer ): LIST
    { L — список типа LIST длиной n.
      Предполагается, что л является степенью числа 2 }
    var
        L1, L2: LIST
    begin
        if л = 1 then
```

```
            return(L);
        else begin
            разбиение L на две части L1 и L2, каждая длиной n/2;
            return(merge(mergesort(L1,n/2), (mergesort(L2,n/2)));
        end
    end; { mergesort }
```

Обозначим через $T(n)$ время выполнения процедуры *mergesort* в самом худшем случае. Анализируя листинг 9.1, можно записать следующее рекуррентное неравенство, ограничивающее сверху $T(n)$:

$$T(n) \le \begin{cases} c_1, & \text{если } n = 1, \\ 2T(n/2) + c_2 n, & \text{если } n > 1. \end{cases} \qquad (9.1)$$

В неравенствах (9.1) константа c_1 соответствует фиксированному количеству шагов, выполняемых алгоритмом над списком L длиной 1. Если $n > 1$, время выполнения процедуры *mergesort* можно разбить на две части. Первая часть состоит из проверки того, что $n \ne 1$, разбиения списка L на две равные части и вызова процедуры *merge*. Эти три операции требуют или фиксированного времени (проверка $n \ne 1$), или пропорционального n — разбиение списка L и выполнение процедуры *merge*. Поэтому можно выбрать такую константу c_2, что время выполнения этой части процедуры *mergesort* будет ограничено величиной $c_2 n$. Вторая часть процедуры *mergesort* состоит из двух рекурсивных вызовов этой процедуры для списков длины $n/2$, которые требуют времени $2T(n/2)$. Отсюда получаем второе неравенство (9.1).

Отметим, что соотношение (9.1) применимо только тогда, когда n четно. Следовательно, формулу для верхней границы $T(n)$ в *замкнутой форме* (т.е. в таком виде, когда формула для $T(n)$ не включает никаких выражений $T(m)$ для $m < n$) можно получить только в случае, если n является степенью числа 2. Но если известна формула для $T(n)$ тогда, когда n является степенью числа 2, то можно оценить $T(n)$ для любых n. Например, для практически всех алгоритмов можно предполагать, что значение $T(n)$ заключено между $T(2^i)$ и $T(2^{i+1})$, если n лежит между 2^i и 2^{i+1}. Более того, нетрудно показать, что в (9.1) выражение $2T(n)$ можно заменить на $T((n+1)/2) + T((n-1)/2)$ для нечетных $n > 1$. Таким образом можно найти решение рекуррентного соотношения в замкнутой форме для любых n.

9.3. Решение рекуррентных соотношений

Существуют три различных подхода к решению рекуррентных соотношений.

1. Нахождение функции $f(n)$, которая мажорировала бы $T(n)$ для всех значений n (т.е. для всех $n > 1$ должно выполняться неравенство $T(n) \le f(n)$). Иногда сначала мы будем определять только вид функции $f(n)$, предполагая, что она зависит от некоторых пока неопределенных параметров (например, $f(n) = an^2$, где a — неопределенный параметр), затем подбираются такие значения параметров, чтобы для всех значений n выполнялось неравенство $T(n) \le f(n)$.
2. В рекуррентном соотношении в правую часть последовательно подставляются выражения для $T(m)$, $m < n$, так, чтобы исключить из правой части все выражения $T(m)$ для $m > 1$, оставляя только $T(1)$. Поскольку $T(1)$ всегда является константой, то в результате получим формулу для $T(n)$, содержащую только n и константы. Такая формула и называется "замкнутой формой" для $T(n)$.
3. Третий подход заключается в использовании общих решений определенных рекуррентных соотношений, приведенных в этой главе или взятых из других источников (см. библиографические примечания).

В этом разделе рассмотрим первых два подхода.

Оценка решений рекуррентных соотношений

Пример 9.1. Рассмотрим первый описанный выше подход на примере соотношения (9.1). Предположим, что $T(n)$ — $an \log n$, где a — пока не определенный параметр. Подставляя $n = 1$, видим, что эта оценка "не работает", так как при $n = 1$ выражение $an \log n$ равно 0, независимо от значения a. Попробуем применить другую функцию: $T(n) = an \log n + b$. При $n = 1$ эта оценка "работает", если положить $b > c_1$.

В соответствии с методом математической индукции предполагаем, что для всех $k < n$ выполняется неравенство

$$T(k) < ak \log k + b, \tag{9.2}$$

и попытаемся доказать, что $T(n) < an \log n + b$.

Пусть $n > 2$. Из неравенств (9.1) имеем $T(n) < 2T(n/2) + c_2 n$. Полагая $k = n/2$, используем в последнем неравенстве оценку (9.2). Получаем

$$T(n) < 2\left(a\frac{n}{2}\log\frac{n}{2}+b\right)+c_2 n =$$
$$= an\log n - an + c_2 n + 2b < an\log n + b. \tag{9.3}$$

Последнее неравенство получено в предположении, что $a > c_2 + b$.

Таким образом, видим, что будет справедлива оценка $T(n) < an \log n + b$, если будут выполняться неравенства $b > c_1$ и $a > c_2 + b$. В данном случае можно удовлетворить этим неравенствам, если положить $b = c_1$ и $a = c_1 + c_2$. Отсюда мы заключаем, что для всех $n > 1$ выполняется неравенство

$$T(n) < (c_1 + c_2)n \log n + c_1. \tag{9.4}$$

Другими словами, $T(n)$ имеет порядок $O(n \log n)$. □

Исходя из рассмотренного примера, сделаем два замечания. Во-первых, если мы предполагаем, что $T(n)$ имеет порядок $O(f(n))$, но по индукции мы не можем доказать неравенства $T(n) < cf(n)$, то это еще не значит, что $T(n) \neq O(f(n))$. Возможно, надо просто добавить константу к функции $cf(n)$, например можно попытаться доказать неравенство $T(n) < cf(n) - 1$!

Во-вторых, мы не определили точной асимптотической степени роста оценочной функции $f(n)$, мы только показали, что она растет не быстрее, чем $O(n \log n)$. Если мы возьмем в качестве оценки более медленно растущие функции, например, такие как $f(n) = an$ или $f(n) = an \log \log n$, то не сможем доказать, что $T(n) < f(n)$. Но в чем причина этого: неверный метод доказательства, или для данного алгоритма в принципе невозможна оценочная функция с меньшей степенью роста? Чтобы ответить на этот вопрос, надо подробнее проанализировать исследуемый алгоритм и рассмотреть время выполнения в самом худшем случае. Для нашей процедуры ***mergesort*** надо показать, что действительно время выполнения равно $\Omega(n \log n)$. Фактически мы показали, что время выполнения процедуры не превосходит $cn \log n$ для любых входных данных, но не для "самых плохих" входных данных. Случай возможных "самых плохих" входных данных мы оставляем в качестве упражнения.

Метод получения оценки решения рекуррентных соотношений, проиллюстрированный в примере 9.1, можно обобщить на некоторые другие функции, ограничивающие сверху время выполнения алгоритмов. Пусть имеется рекуррентное соотношение

$$T(1) = c,$$
$$T(n) < g(T(n/2), n) \text{ для } n > 1. \tag{9.5}$$

Отметим, что соотношение (9.5) обобщает соотношение (9.1), которое получается из (9.5), если положить $g(x, y) = 2x + c_2 y$. Конечно, возможны еще более общие соотношения, чем (9.5). Например, функция g может включать в себя все значения $T(n - 1)$, $T(n - 2)$, ..., $T(1)$, а не только $T(n/2)$. Также могут быть заданы значения для $T(1)$, $T(2)$, ..., $T(k)$ и рекуррентное соотношение, которое применимо только для $n > k$. Читатель может самостоятельно попробовать приме-

нить к этим обобщенным рекуррентным соотношениям описанный выше метод оценки решения этих соотношений.

Вернемся к соотношению (9.5). Предположим, что выбранная оценочная функция $f(a_1, .., a_j,$ га) зависит от параметров $a_1, ..., a_j$, нам надо доказать индукцией по n, что $T(n) < f(a_1, ..., a_j, n)$. (В примере 9.1 $f(a_1, a_2,$ га$) = a_1 n \log n + a_2$, параметры a_1 и a_2 обозначены как а и b.) Для того чтобы оценка $T(n) \leq f(a_1, ..., a_j,$ га) была справедливой для всех $n > 1$, надо найти такие значения параметров $a_1, ..., a_j$, чтобы выполнялись неравенства

$$f(a_1, ..., a_j, 1) \geq c,$$
$$f(a_1 ..... a_j, \text{га}) > g(f(a_1,..., a_j, \quad \text{га}/2), \quad \text{га}). \qquad (9.6)$$

В соответствии с методом математической индукции можно подставить функцию / вместо T в правую часть неравенства (9.5). Получаем

$$T(n) < g(f(a_1,..., a_j, \quad \text{га}/2), \quad \text{га}). \qquad (9.7)$$

Если неравенство (9.6) выполняется (уже подобраны соответствующие значения параметров), то, применив его в (9.7), будем иметь то, что требуется доказать: $T(n) < f(a_1, ..., a_j,$ га).

В примере 9.1 мы имели $g(x, y) = 2x + c_2 y$ и $f(a_1, a_2,$ га$) = a_1 n \log n + a_2$. Параметры a_1 и a_2 надо подобрать так, чтобы выполнялись неравенства

$$f(a_1, a_2, 1) > c_1,$$
$$f(a_1, a_2, \text{га}) = a_1 n \log\text{га} + a_2 > 2\left(a_1 \frac{n}{2} \log \frac{n}{2} + a_2\right) + c_2 n.$$

Для этого достаточно положить $a_2 = c_1$ и $a_1 = c_1 + c_2$.

Оценка решения рекуррентного соотношения методом подстановки

Если мы не знаем вида оценочной функции или не уверены в том, что выбранная оценочная функция будет наилучшей границей для $T(n)$, то можно применить подход, который в принципе всегда позволяет получить точное решение для $T(n)$, хотя на практике он часто приводит к решению в виде достаточно сложных сумм, от которых не всегда удается освободиться. Рассмотрим этот подход на примере рекуррентного соотношения (9.1). Заменим в этом соотношении га на га/2, получим

$$T(n/2) < 2T(n/4) + c_2 n/2. \qquad (9.8)$$

Подставляя правую часть неравенства (9.8) вместо $T(n/2)$ в неравенство (9.1), будем иметь

$$T(n) < 2(2T(n/4) + c_2 n/2) + c_2 n = 4T(n/4) + 2c_2 n. \qquad (9.9)$$

Аналогично, заменяя в неравенстве (9.1) га на га/4, получаем оценку для $T(n/4)$: $T(n/4) < 2T(n/8) + c_2 n/4$. Подставляя эту оценку в неравенство (9.9), получаем

$$T(n) < 8T(n/8) + 3c_2 n. \qquad (9.10)$$

Надеемся, читатель понял принцип подстановки. Индукцией по i для любого i можно получить следующее соотношение:

$$T(n) < 2^i T(n/2^i) + i c_2 n. \qquad (9.11)$$

Предположим, что га является степенью числа 2, например га = 2*. Тогда при i = k в правой части неравенства (9.11) будет стоять $T(1)$:

$$T(n) < 2^k T(1) + k c_2 n. \qquad (9.12)$$

Поскольку га = 2^k, то $k = \log$га. Так как $T(1) < c_1$, то из (9.12) следует, что

$$T(n) < c_1 n + c_2 n \log\text{га}. \qquad (9.13)$$

Неравенство (9.13) показывает верхнюю границу для $T(n)$, это и доказывает, что $T(n)$ имеет порядок роста не более $O(n \log n)$.

9.4. Общее решение большого класса рекуррентных уравнений

Рассмотрим решение рекуррентных соотношений, когда исходную задачу размера *п* можно разделить на *а* подзадач каждую размера *п/Ь.* Для определенности будем считать, что выполнение задачи размера 1 требует одну единицу времени и что время "сборки" подзадач, составляющих исходную задачу размера *п,* требует $d(n)$ единиц времени. В примере процедуры ***mergesort*** $a = Ь = 2$ и $d(n) = c_2n/c_1$ в единицах c_1. Тогда, если обозначить через $T(n)$ время выполнения задачи размера п, будем иметь

$$T(1) = 1,$$
$$T(n) = aT(n/b) + d(n). \qquad (9.14)$$

Отметим, что соотношение (9.14) можно применить только тогда, когда га является целой степенью числа *Ь.* Но если функция *Т(п)* достаточно гладкая, то решение, полученное в предположении, что га является целой степенью числа *Ь,* можно будет распространить на другие значения *п.*

Заметим также, что в (9.14) мы имеем уравнение, тогда как в (9.1) имели неравенство. Причина заключается в том, что в (9.14) используется пока произвольная функция *d(n),* поэтому мы имеем право записать точное равенство. А в (9.1) выражение c_2n соответствует времени выполнения операции слияния списков в самом худшем случае и поэтому является верхней оценкой; фактическое время выполнения процедуры *mergesort* на входных данных размера га может быть меньше $2T(n/2) + c_2n$. С другой стороны, не имеет значения, используем ли мы в рекуррентных соотношениях знак равенства или знак неравенства, поскольку мы ищем только верхнюю границу времени выполнения в самом худшем случае.

Для решения уравнения (9.14) применим метод подстановки, рассмотренный в предыдущем разделе. Итак, подставим в (9.14) вместо га выражение n/b^i:

$$T\left(\frac{n}{b^i}\right) = aT\left(\frac{n}{b^{i+1}}\right) + d\left(\frac{n}{b^i}\right). \qquad \mathbf{(9.15)}$$

Подставляя в (9.14) равенства (9.15) последовательно для *i* — 1, 2, ..., получим

$$ТЫ - aT\left(\frac{n}{b}\right) + d(n) = a\left(aT\left(\frac{n}{b^2}\right) + d\left(\frac{n}{b}\right)\right) + d(n) \ , \ a^2T\left(\frac{n}{b^2}\right) + ad\left(\frac{n}{b}\right) + d(n) =$$
$$= a^2\left(aT\left(\frac{n}{b^3}\right) + d\left(\frac{n}{b^2}\right)\right) + ad\left(\frac{n}{b}\right) + \ . = \ a^3T\left(\frac{n}{b^3}\right) + a^2d\left(\frac{n}{b^2}\right) + ad\left(\frac{n}{b}\right) + d(n) =$$
$$= \ldots = a^iT\left(\frac{n}{b^i}\right) + \sum_{j=0}^{i-1} a^j d\left(\frac{n}{b^j}\right).$$

Если, как мы предположили, га $= b^k$, тогда при $i = k$ $T(n/b^k) = T(1) = 1$ и мы получаем формулу

$$T(n) - a^k + \sum_{j=0}^{k-1} a^j d(b^{k-j}). \qquad (9.16)$$

Так как $k = \log_b n$, то первое выражение в (9.16) можно записать как $a^{\log_b n}$ или, что эквивалентно, как га'ofi'o. Таким образом, это выражение обозначает га в степени, которая зависит от а и b. Например, в случае процедуры *mergesort a — b = 2,* поэтому первое выражение равно просто га. В общем **случае,** чем больше *а* (т.е. надо решить большее количество подзадач), тем больший показатель степени га. При больших значениях b (чем больше b, тем меньше размер подзадач) показатель степени га будет меньше.

Однородные и частные решения

Интересно исследовать, какова роль обоих выражений в формуле (9.16). Первое выражение a* или ra[1]"*"" называется *однородным решением* (по аналогии с дифференциальными уравнениями). Однородное решение — это точное решение уравнения (9.14), когда функция $d(n)$, называемая *управляющей функцией,* равна 0 для всех n. Другими словами, однородное решение соответствует стоимости выполнения всех подзадач, если их можно объединить "бесплатно".

С другой стороны, второе выражение формулы (9.16) соответствует стоимости создания подзадач и комбинирования их результатов. Назовем это выражение *частным решением* (также по аналогии с теорией дифференциальных уравнений). Частное решение зависит как от управляющей функции, так и от количества и размера подзадач. Существует практическое правило: если однородное решение больше управляющей функции, то частное решение имеет тот же порядок роста, что и однородное решение. Если же управляющая функция растет на порядок n^{ε} (для некоторого e > 0) быстрее однородного решения, тогда частное решение имеет такой же порядок роста, как и управляющая функция. Когда управляющая функция имеет с однородным решением одинаковый порядок роста или растет не медленнее, чем $\log^{k} n$ для некоторого k, тогда частное решение растет как управляющая функция, умноженная на logл.

При исследовании реализации любого алгоритма важно распознать, когда однородное решение будет больше управляющей функции, так как в этом случае любой более быстрый способ объединения подзадач не окажет существенного влияния на эффективность всего алгоритма. В этой ситуации для ускорения выполнения алгоритма надо или уменьшить количество подзадач, или уменьшить их размер. Только это может привести к тому, что однородное решение будет меньше общего времени выполнения алгоритма.

Если управляющая функция превышает однородное решение, тогда надо попытаться уменьшить эту функцию. Например, в случае процедуры **mergesort,** где $a = b = 2$ и $d(n) = cn$, мы видели, что частное решение имеет порядок $O(n \log n)$. Если сделать функцию $d(n)$ растущей медленнее, чем линейная, например как n , тогда, как мы увидим далее, частное решение будет расти медленнее, чем линейная функция, и общее время выполнения алгоритма будет иметь порядок $O(n)$, такой же, как и однородное решение.[1]

Мультипликативные функции

Частное решение в формуле (9.16) оценить достаточно трудно, даже если известен вид функции $d(n)$. Но для определенного, достаточно широкого класса функций $d(n)$ можно найти точное решение (9.16). Будем говорить, что функция f целочисленного аргумента называется *мультипликативной,* если для всех положительных целых чисел x и y справедливо равенство $f(xy) - f(x)f(y)$.

Пример 9.2. Для нас наибольший интерес представляют мультипликативные функции вида n^{α}, где а — некоторая положительная константа. Чтобы показать, что функция $f(n) = n^{\alpha}$ является мультипликативной, достаточно заметить, что $(xy)^{\alpha} = x^{\alpha}y^{\alpha}$. □

Пусть в (9.16) функция $d(n)$ является мультипликативной, тогда $d(b\ ') = (d(b))\ '$. В этом случае частное решение запишется в следующем виде:

$$\sum_{j=0}^{k-1} a^j (d(b))^{k-j} = d(b)^k \sum_{j=0}^{k-1} \left(\frac{a}{d(b)} \right)^j = d(b)^k \frac{\left(\frac{a}{d(b)} \right)^k - 1}{\frac{a}{d(b)} - 1} = \frac{a^k - d(b)^k}{\frac{a}{d(b)} - 1}. \tag{9.17}$$

---

[1] Но в случае процедуры *mergesort* нельзя надеяться, что найдется способ слияния двух списков из $n/2$ элементов за время, меньшее по порядку, чем линейное, поскольку здесь в любом случае необходимо просмотреть все n элементов.

Теперь необходимо рассмотреть три ситуации, зависящие от того, будет ли параметр a больше, меньше или равен $d(b)$.

1. Если $a > d(b)$, тогда последнее выражение в (9.17) имеет порядок $O(a^k)$, который можно переписать как га'"*"'[1], поскольку $k = \log_b n$. В этом случае частное и однородные решения имеют одинаковый порядок роста, который зависит только от a и Ъ, но не от управляющей функции. Поэтому в данной ситуации уменьшение времени выполнения алгоритма можно достичь путем уменьшения a или увеличения b, уменьшение порядка роста функции $d(n)$ здесь не поможет.
2. Если а < $d(b)$, тогда последнее выражение в (9.17) имеет порядок $O(d(b)^k)$, или, что то же самое, $O(n^{\log_b d(b)})$. В этом случае частное решение превышает однородное. Поэтому для уменьшения времени выполнения алгоритма можно манипулировать как функцией $d(n)$, так и параметрами a и b. В важном частном случае, когда $d(n)$ =га™,$d(b)$ будет равно b^α и $\log_b(b^\alpha)$= а. Тогда частное решение имеет порядок $O(n^\alpha)$ или $O(d(n))$.
3. Если $a = d(b)$, тогда надо пересмотреть решение (9.17), поскольку в данном случае нельзя применять формулу суммирования членов геометрической прогрессии. В этой ситуации суммируем следующим образом:

$$\sum_{j=0}^{k-1} a^j (d(b))^{k-j} = d(b)^k \sum_{j=0}^{k-1} \left(\frac{a}{d(b)} \right)^j = d(b)^k \sum_{j=0}^{k-1} 1 = d(b)^k k = n^{\log_b d(b)} \log_b n. \qquad (9.18)$$

Поскольку $a = d(b)$, то частное решение (9.18) равно однородному решению, умноженному на $\log_b n$. Здесь опять частное решение превосходит однородное. В частном случае, когда $d(n) = n^\alpha$, решение (9.18) имеет порядок $O(n^\alpha \log n)$.

Пример 9.3. Рассмотрим следующие рекуррентные уравнения с начальным значением $T(1) = 1$.

1. $T(n) = 4T(n/2) + n$.
2. $T(n) = 4T(n/2) + n^2$.
3. $T(n) = 4T(n/2)+$ ге^3.

В каждом уравнении $a = 4$ и $b = 2$, следовательно, однородное решение равно га^2. В уравнении (1) $d(n)$ — n, поэтому $d(b) = 2$. Поскольку $a = 4 > d(b)$, то частное решение также имеет порядок га^2. Поэтому здесь $T(n) = O(n^2)$.

В уравнении (3) $d(n)$ = га^2, $d(b) = 8$ и а < $d(b)$. Тогда частное решение имеет порядок $O(n^{\log_b d(b)}) = O(n^3)$ и $T(n)$ в этом уравнении также равно $O(n^3)$.

В уравнении (2) $d(b) = 4 = a$, поэтому решение дается формулой (9.18). Здесь как частное решение, так и $T(n)$ имеют порядок $O(n^2 \log n)$. П

Другие управляющие функции

Существуют другие типы управляющих функций, отличные от мультипликативных, которые позволяют получить замкнутую форму решения (9.16). Мы рассмотрим только два примера таких функций. В первом примере получаем небольшое обобщение мультипликативных функций, а именно, здесь рассматриваются мультипликативные функции, умноженные на константу, которая больше или равна единице. Во втором примере просто показан частный случай функции $d(n)$, позволяющий получить замкнутую форму решения (9.16).

Пример 9.4. Рассмотрим следующее рекуррентное уравнение:

$$T(1) = 1,$$
$$T(n) = 3T(n/2) + 2n^{1.5}.$$

Выражение $2n^{1.5}$ не является мультипликативной функцией, но функция $n^{1.5}$ является. Сделаем замену *Щn)* = *T(n)/2* для всех л. Тогда исходное рекуррентное перепишется в виде

$$U(1) = 1/2,$$
$$Щn) = 3U(n/2) + n^{1.5}.$$

Если бы $U(1)$ равнялось 1, тогда однородное решение равнялось бы $n^{\log 3} < n^{1.59}$. Для $U(1)$ — 1/2 можно показать, что однородное решение не больше $n^{1.59}/2$, т.е. имеет порядок $O(n^{1.59})$. На частное решение значение $U(1)$ не влияет. Поскольку в данном случае $a = 3$, $Ъ = 2$ и $b^{1.5} = 2.83 < a$, то частное решение, а также и *Щn)* имеют порядок роста $O(n^{1.59})$. Так как $T(n)$ — $2U(n)$, то $T(n)$ тоже имеет порядок $O(n^{1.59})$, точнее $O(n^{\log 3})$. П

Пример 9.5. Рассмотрим рекуррентное уравнение

$$T(1) = 1,$$
$$T(n) = 2T(n/2) + л \log n.$$

Легко проверить, что в данном случае однородное решение равно л, поскольку а = Ъ = 2. Но функция *d(n)* — *n* $\log n$ мультипликативной не является, поэтому надо попытаться найти значение суммы в формуле (9.16). В данном случае имеем

$$\sum_{j=0}^{k-1} 2^j 2^{k-j} \log(2^{k-j}) = 2^k \sum_{j=0}^{k-1} (k-j) = 2^{k-1} k(k+1).$$

Так как ft = $\log n$, то частное решение имеет порядок $O(n \log^2 n)$. Поскольку здесь частное решение больше однородного, то отсюда следует, что *T(n)* также будет иметь порядок $O(n \log^2 n)$. П

Упражнения

9.1. Запишите рекуррентное соотношение для времени выполнения следующего алгоритма, предполагая, что *n* является степенью числа 2.

```
function path ( s/ t, n: integer ): boolean;
    begin
        if n = 1 then
            if edge(s, t) then
                return(true)
            else
                return(false);
        for i:= 1 to n do
            if path(s,i,n div 2) and path(i,t,n div 2) then
                return(true);
        return(false)
    end; { path }
```

Функция *edge(i, j)* возвращает значение **true**, если вершины i и у в графе с л вершинами соединены ребром либо *i = j*, и значение false — в противном случае. Что делает функция *path* (путь)?

9.2. Решите следующие рекуррентные уравнения, если $T(1) = 1$.

а) $T(n) = 3T(n/2) +$ л;

б) $T(n) = 3T(n/2) + n^2$;

в) $T(n) = 8T(n/2) + n^3$.

9.3. Решите следующие рекуррентные уравнения, если $T(1) = 1$.

а) $T(n) = 4T(n/2) +$ п;

б) $T(n) = 4T(n/2) + n^2$;

в) $T(n) = 9T(n/2) + n^3$.

9.4. Найдите верхнюю оценку для $T(n)$, удовлетворяющих следующим рекуррентным уравнениям и предположению, что $T(1) = 1$.

а) $T(n)= T(n/2) + 1$;

б) $T(n) = 2T(n/2)+ \log n$;

в) $T(n) = 2T(n/2)+ n$;

г) $T(n) = 2T(n/2)+ n^2$.

*9.5. Найдите верхнюю оценку для $T(n)$, удовлетворяющих следующим рекуррентным соотношениям:

а) $T(1) = 2$,

$T(n) = 2T(n - 1) + 1$ при л > 2.

б) Г(1) = 1,

$T(n)= 2T(n - 1) +$ га при л > 2.

9.6. Проверьте ответы упражнения 9.5, решив рекуррентные соотношения методом подстановок.

9.7. Обобщите упражнение 9.6 для решения произвольных рекуррентных уравнений вида

$T(1) = 1$,

$T(n)= aT(n - 1) + d(n)$ при $n \geq 2$

в терминах параметра а и функции $d(n)$.

*9.8. В упражнении 9.7 положите $d(n) = c^n$ для некоторой константы $c > 1$. Как в этом случае решение $T(n)$ будет зависеть от соотношения а и с? Какой вид решения $T(n)$?

**9.9. Найдите $T(n)$, если

$T(1) - 1$,

$T(n) = \sqrt{n}T(\sqrt{n})$ + л при л > 2.

9.10. Найдите замкнутые выражения для следующих сумм:

а) $\sum_{i=0}^{n} i$, б) $\sum_{i=0}^{n} i^k$, в) $\sum_{i=0}^{n} 2^i$, г) $\sum_{i=0}^{n} C_n^i$.

*9.11. Покажите, что число различных порядков, в соответствии с которыми можно перемножить последовательность из л матриц, удовлетворяет следующим рекуррентным соотношениям:

$T(1) = 1$,

$$T(n) = \sum_{i=1}^{n-1} T(i)T(n-i).$$

Докажите, что $T(n+1) = \frac{1}{л+1}$,С," . Числа $T(n)$ называются *числами Каталана*.

**9.12. Покажите, что число сравнений $T(n)$, необходимое для сортировки л элементов посредством метода сортировки слиянием, удовлетворяет соотношениям

$T(1) = $ О,

$T(n)= T([n/2]) + T(\lceil n/2 \rceil)$ + л $- 1$,

где $[x]$ обозначает целую часть числа x, а Ы — наименьшее целое, большее или равное x. Докажите, что решение этих рекуррентных соотношений имеет вид

$$T(n) = n\lceil \log n \rceil - 2^{\lceil \log n \rceil} + 1.$$

9.13. Покажите, что число булевых функций п переменных удовлетворяет рекуррентным соотношениям

$T(1) = 4,$

$T(n) = (T(n - 1))^2.$

Найдите $T(n)$.

****9.14.** Покажите, что число двоичных деревьев высотой не более n удовлетворяет рекуррентным соотношениям

$T(1) - 1,$

$T(n) - (T(n - 1))^2 + 1.$

Докажите, что $T(n) = \lceil k^{2^n} \rfloor$ для некоторой константы k. Каково значение ft?

Библиографические примечания

В работах [9], [44], [70] и [71] можно найти дополнительный материал по решению рекуррентных соотношений. В [4] показано, что многие нелинейные рекуррентные уравнения вида $T(n) = (T(n - 1))^2 + g(n)$ имеют решения в форме $T(n) = \lceil ft^{2^n} \rfloor$, где k — константа, как в упражнении 9.14.

ГЛАВА 10

Методы разработки алгоритмов

К настоящему времени специалисты по вычислительной технике разработали ряд эффективных методов, которые нередко позволяют получать эффективные алгоритмы решения больших классов задач. Некоторые из наиболее важных методов, такие как "разделяй и властвуй" (декомпозиции), динамическое программирование, "жадные" методы, поиск с возвратом и локальный поиск, представлены в этой главе. Пытаясь разработать алгоритм решения той или иной задачи, часто бывает полезно задать вопрос: "Какой тип решения позволяет получить метод декомпозиции, динамическое программирование, "жадный" метод или какой-либо другой стандартный метод?"

Следует, однако, подчеркнуть, что существуют задачи, например NP-полные задачи, для которых эти и любые другие известные методы не обеспечивают эффективных решений. Когда встречается подобная задача, нередко бывает полезно определить, обладают ли входные данные для этой задачи какими-либо особыми характеристиками, которыми можно воспользоваться при попытках найти требуемое решение, и можно ли воспользоваться каким-либо приближенным решением (если такое решение легко получить) вместо точного решения, получить которое бывает очень трудно.

10.1. Алгоритмы „разделяй и властвуй"

Возможно, самым важным и наиболее широко применимым методом проектирования эффективных алгоритмов является метод, называемый *методом декомпозиции* (или метод "разделяй и властвуй", или метод разбиения). Этот метод предполагает такую декомпозицию (разбиение) задачи размера n на более мелкие задачи, что на основе решений этих более мелких задач можно легко получить решение исходной задачи. Мы уже знакомы с рядом применений этого метода, например в сортировке слиянием или в деревьях двоичного поиска.

Чтобы проиллюстрировать этот метод, рассмотрим хорошо известную головоломку "Ханойские башни". Имеются три стержня A, B и C. Вначале на стержень A нанизаны несколько дисков: диск наибольшего диаметра находится внизу, а выше — диски последовательно уменьшающегося диаметра, как показано на рис. 10.1. Цель головоломки — перемещать диски (по одному) со стержня на стержень так, чтобы диск большего диаметра никогда не размещался выше диска меньшего диаметра и чтобы в конце концов все диски оказались нанизанными на стержень B. Стержень C можно использовать для временного хранения дисков.

Для решения этой головоломки подходит следующий простой алгоритм. Представьте, что стержни являются вершинами треугольника. Пронумеруем все перемещения дисков. Тогда при перемещениях с нечетными номерами наименьший диск нужно перемещать в треугольнике на соседний стержень по часовой стрелке. При перемещениях с четными номерами выполняются другие допустимые перемещения, не связанные с наименьшим диском.

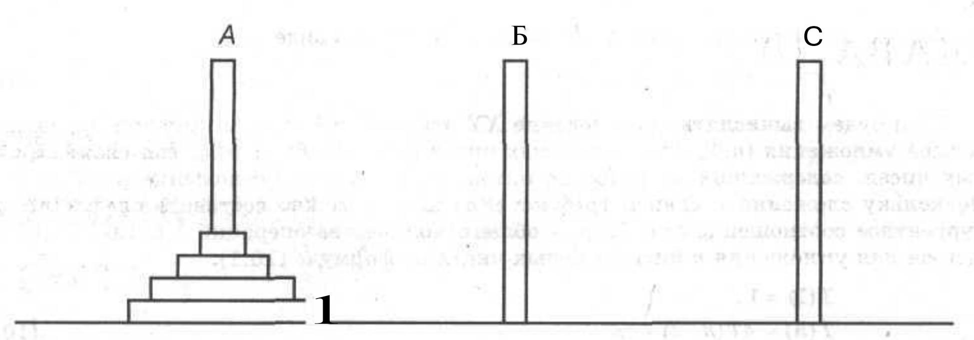


Рис. 10.1. Исходное положение в головоломке "Ханойские башни"

Описанный выше алгоритм, конечно, правильный и, к тому же, весьма лаконичный, правда, нелегко понять, почему он "работает", да и вряд ли такой алгоритм быстро придет вам в голову. Попробуем вместо этого применить метод декомпозиции. Задачу перемещения n наименьших дисков со стержня A на стержень B можно представить себе состоящей из двух подзадач размера $n — 1$. Сначала нужно переместить $n — 1$ наименьших дисков со стержня А на стержень С, оставив на стержне A n-й наибольший диск. Затем этот диск нужно переместить с А на В. Потом следует переместить n - 1 дисков со стержня C на стержень B. Это перемещение га - 1 дисков выполняется путем рекурсивного применения указанного метода.[1] Поскольку диски, участвующие в перемещениях, по размеру меньше тех, которые в перемещении не участвуют, не нужно задумываться над тем, что находится под перемещаемыми дисками на стержнях А, B или С. Хотя фактическое перемещение отдельных дисков не столь очевидно, а моделирование вручную выполнить непросто из-за образования стеков рекурсивных вызовов, с концептуальной точки зрения этот алгоритм все же довольно прост для понимания и доказательства его правильности (а если говорить о быстроте разработки, то ему вообще нет равных). Именно легкость разработки алгоритмов по методу декомпозиции обусловила огромную популярность этого метода; к тому же, во многих случаях эти алгоритмы оказываются более эффективными, чем алгоритмы, разработанные Традиционными методами.[2]

Умножение длинных целочисленных значений

Рассмотрим задачу умножения двух n-битовых целых чисел X и Y. Вспомним, что алгоритм умножения га-битовых (или га-разрядных) целых чисел, изучаемый в средней школе, связан с вычислением n промежуточных произведений размера n и поэтому является алгоритмом $O(n^2)$, если на каждом шаге выполняется лишь умножение или сложение одного бита или разряда. Один из вариантов метода декомпозиции применительно к умножению целых чисел заключается в разбиении каждого из чисел X и Y на два целых числа по га/2 битов в каждом, как показано на рис. 10.2. (Для простоты в данном случае предполагается, что га является степенью числа 2.)

X:=	A	B	$X = A2^{n/2} + B$
Y:=	C	D	$Y = C2^{n/2} + D$

Рис. 10.2. Разбиение n-битовых целых чисел на n/2-битовые составляющие

---

[1] Чтобы прояснить идею алгоритма, опишем еще один шаг решения головоломки. После того как n - 1 дисков перемещены на стержень C, а наибольший диск помещен на стержень B, n - 2 наименьших диска со стержня C перемещаются на стержень А и (л — 1)-й диск переносится на диск B. Далее решается задача с n - 2 дисками, находящимися на диске А. — *Прим. ред.*

[2] В случае "Ханойских башен" алгоритм декомпозиции на самом деле ничем не отличается от того алгоритма, который был описан вначале.

Теперь произведение чисел X и Y можно записать в виде

$$XY = AC2^n + (AD + BC)2^{n/2} + BD. \quad (10.1)$$

Если будем вычислять произведение XY этим способом, нам придется выполнить четыре умножения (л/2)-битовых целых чисел *(AC, AD, EC* и *BD),* три сложения целых чисел, содержащих не более 2л битов, и два сдвига (умножение на 2" и $2^{n/2}$). Поскольку сложения и сдвиги требуют $O(n)$ шагов, можно составить следующее рекуррентное соотношение для $T(n)$ — общего количества операций с битами, требующегося для умножения n-битных целых чисел по формуле (10.1):

$$T(1) = 1,$$
$$T(n) = 4T(n/2) + cn. \quad (10.2)$$

Используя рассуждения, подобные приведенным в примере 9.4, можно обосновать в (10.2) значение константы *c,* равное 1. Тогда управляющая функция $d(n)$ просто равняется n, и однородное и частное решения имеют порядок О(л").

Если формула (10.1) используется для умножения целых чисел, асимптотическая эффективность будет, таким образом, не больше, чем при использовании метода, которому обучают в средней школе. Не следует, однако, забывать, что для уравнений типа (10.2) мы получаем асимптотическое улучшение в случае, если сокращается количество подзадач. Рассмотрим другую формулу для умножения чисел X и У:

$$XT = AC2^n + [(A - B)(D - C) + AC + BD]2^{n/2} + BD. \quad (10.3)$$

Несмотря на то что формула (10.3) выглядит сложнее, чем (10.1), она требует лишь трех умножений (л/2)-битовых целых чисел *(AC, BD* и $(A-B)(D-C)$), шести сложений или вычитаний и двух сдвигов. Поскольку все эти операции, кроме умножений, выполняются за $O(n)$ шагов, время $T(n)$ для умножения л-битовых целых чисел в соответствии с (10.3) задается соотношениями

$$T(1) = 1,$$
$$T(n) = 3T(n/2) + cn.$$

Их решением является $T(n) = O(n^{\log_2 3}) = O(n^{1,59})$.

В листинге 10.3 приведен полный код алгоритма, предусматривающий умножение как отрицательных, так и положительных целых чисел. Обратите внимание, что строки (8) - (11) выполняются путем копирования битов, а умножение на 2" и $2^{n/2}$ в строке (16) — путем сдвига. Отметим также, что в строке (16) результату придается нужный знак.

Листинг 10.1. Алгоритм умножения целых чисел методом декомпозиции

```
        function mult ( X, Y, n: integer ) : integer;
            { X и Y — целые числа со знаком < 2ⁿ. л — степень числа 2.
                Функция возвращает значение произведения XY }
            var
                s: integer; { содержит знак произведения XY }
                m1, m2, m3: integer; { содержат три произведения }
                A, B, C, D: integer; {содержат левые и правые половины X и Y}
            begin
(1)             s:= sign(X) * sign(Y);
(2)             X:= abs(X);
(3)             Y:= abs(Y); { теперь X и Y— положительные числа }
(4)             if л = 1 then
(5)                 if (X = 1) and (Y = 1) then
(6)                     return (s)
                    else
```

```
(7)                 return  (0)
            else begin
(8)                 A:= левые n/2 биты числа X;
(9)                 B:= правые п/2 биты числа X;
(10)                C:= левые л/2 биты числа Y;
(11)                D:= правые л/2 биты числа Y;
(13)          m1:= mult(A, C, n/2);
(14)                m2:= mult(A-B, D-C, n/2);
(15)                m3: = mult(B, D, n/2);
(16)                return (s * (m1*2^n + (ml + m2 + т3) * 2^(n/2) + т3))
            end
        end; { mult }
```

Обратите внимание, что алгоритм, разработанный по методу декомпозиции (листинг 10.1), оказывается асимптотически быстрее, чем метод, **которому обучают** в средней школе (требует только $O(n^{1,59})$ шагов вместо $O(n^2)$). Таким образом, возникает закономерный вопрос: если этот алгоритм настолько лучше, почему именно его не изучают в средней школе? На этот вопрос можно дать два ответа. Прежде всего, этот алгоритм удобен для реализации на компьютере; если бы мы попытались излагать его в средней школе, учащиеся так и не научились бы умножать целые числа. Более того, мы игнорировали константы пропорциональности. В то время как процедура *mult* асимптотически превосходит обычный метод, константы таковы, что в случае небольших задач (реально — до 500 бит) метод, излагаемый в средней школе, оказывается лучше.

Составление графика проведения теннисного турнира

Метод декомпозиции получил широкое применение не только при разработке алгоритмов, но и в проектировании электронных схем, построении математических доказательств и в других сферах. В качестве иллюстрации приведем лишь один пример. Рассмотрим составление расписания проведения теннисного турнира по круговой схеме для n = 2* игроков. Каждый игрок должен сыграть со всеми другими игроками, при этом каждый игрок должен играть по одному матчу в день в течение n — 1 дней — минимального количества дней, необходимых для проведения всего турнира.

Расписание проведения турнира, таким образом, представляет собой таблицу, состоящую из л строк и n — 1 столбцов; элементом на пересечении строки i и столбца j является номер игрока, с которым игрок i должен провести матч в у'-й день.

Метод декомпозиции позволяет составить расписание для половины игроков. Это расписание составляется на основе рекурсивного применения данного алгоритма для половины этой половины игроков и т.д. Когда количество игроков будет сокращено до двух, возникнет "базовая ситуация", в которой мы просто устанавливаем порядок проведения встреч между ними.

Допустим, в турнире участвуют восемь игроков. Расписание для игроков 1 - 4 заполняет верхний левый угол (4 строкиХЗ столбца) составляемого расписания. Нижний левый угол (4 строкиХЗ столбца) этого расписания должен свести между собой игроков с более высокими номерами (5 - 8). Эта часть расписания получается путем прибавления числа 4 к каждому элементу в верхнем левом углу.

Итак, нам удалось упростить задачу. Теперь остается свести между собой игроков с низкими и более высокими номерами. Сделать это нетрудно: надо на 4-й день свести в пары игроков, имеющих номера 1 - 4, с игроками **5 – 8** соответственно, а в последующие дни просто циклически переставлять номера **5** - 8. Этот процесс показан на рис. 10.3. Надеемся, теперь читатель сможет обобщить описанный алгоритм и составить расписание для 2* игроков при любом значении k.

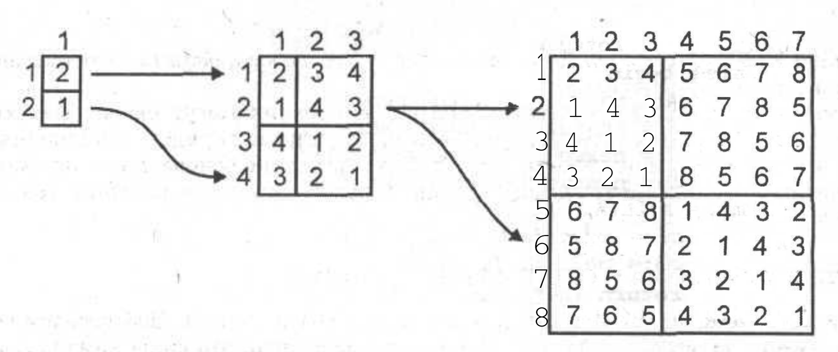


Рис. 10.3. Организация кругового турнира для восьми игроков

Баланс подзадач

При проектировании алгоритмов приходится идти на различные компромиссы. Очевидно, что по мере возможности необходимо сбалансировать вычислительные затраты на выполнение различных частей алгоритма. Например, в главе 5 было показано, что 2-3 дерево позволяет сбалансировать затраты на поиск элементов с затратами на их вставку, в то время как более прямолинейные методы требуют выполнения $O(n)$ шагов как для каждого поиска, так и для каждой вставки (несмотря на то что другие операции можно выполнить за постоянное число шагов).

Аналогично, при использовании алгоритмов декомпозиции желательно, чтобы подзадачи были примерно одинакового размера. Например, сортировку вставками можно рассматривать как разбиение задачи на две подзадачи — одна размером 1, а другая — n - 1, причем максимальные затраты на выполнение слияния равняются n шагам. В результате приходим к рекуррентному соотношению $T(n) = T(1) + T(n - 1) + n$, которое имеет решение $O(n^2)$. В то же время сортировка слиянием разбивает задачу на две подзадачи, каждая размером $n/2$, а ее эффективность равняется $O(n \log n)$. Складывается впечатление, что разбиение задачи на равные (или примерно равные) подзадачи является важным фактором обеспечения высокой эффективности алгоритмов.

10.2. Динамическое программирование

Нередко не удается разбить задачу на небольшое число подзадач, объединение решений которых позволяет получить решение исходной задачи. В таких случаях мы можем попытаться разделить задачу на столько подзадач, сколько необходимо, затем каждую подзадачу разделить на еще более мелкие подзадачи и т.д. Если бы весь алгоритм сводился именно к такой последовательности действий, мы бы получили в результате алгоритм с экспоненциальным временем выполнения.

Но зачастую удается получить лишь полиномиальное число подзадач и поэтому ту или иную подзадачу приходится решать многократно. Если бы вместо этого мы отслеживали решения каждой решенной подзадачи и просто отыскивали в случае необходимости соответствующее решение, мы бы получили алгоритм с полиномиальным временем выполнения.

С точки зрения реализации иногда бывает проще создать таблицу решений всех подзадач, которые нам когда-либо придется решать. Мы заполняем эту таблицу независимо от того, нужна ли нам на самом деле конкретная подзадача для получения общего решения. Заполнение таблицы подзадач для получения решения определен-

ной задачи получило название *динамического программирования* (это название происходит из теории управления).[1]

Формы алгоритма динамического программирования могут быть разными — общей их "темой" является лишь заполняемая таблица и порядок заполнения ее элементов. Соответствующие методы проиллюстрируем на основе двух примеров: вычисление шансов на победу команд в спортивных турнирах и решение так называемой "задачи триангуляции".

Вероятность победы в спортивных турнирах

Допустим, две команды *Аи В* проводят между собой турнир. Победителем считается тот, кто первым выиграет n матчей. Можно предположить, что силы команд А и Б примерно равны, т.е. у каждой из них есть 50% шансов выиграть очередной матч. Допустим, $P(i, j)$ — вероятность того, что если A для победы нужно провести i матчей, *а В — j* матчей, то в конечном счете победу на турнире одержит *А*. Например, если в турнире "Мировой серии" команда Dodgers выиграла два матча, а команда Yankees — один, то $i = 2, j = 3$ и оказывается (мы в этом еще убедимся), что $P(2, 3)$ равняется 11/16.

Чтобы вычислить $P(i, j)$, можно воспользоваться рекуррентным уравнением с двумя переменными. Во-первых, если $i = 0$ и $j > 0$, то команда A уже выиграла турнир, поэтому $P(0, j) = 1$. Аналогично, $P(i, 0) = 0$ для i > 0. Если как i, так и j больше нуля, командам придется провести по крайней мере еще один матч с равными шансами выиграть эту игру. Таким образом, $P(i, j)$ должно быть средним значением $P(i - 1, j)$ и $P(l, j - 1)$, первое из этих выражений представляет собой вероятность того, что команда A выиграет турнир, если победит в следующем матче, а второе — вероятность того, что команда A выиграет турнир, даже если проиграет следующий матч. Итак, получаем следующие рекуррентные соотношения:

$P(i, j) = 1$, если i = 0 и $j > O$,

$P(i,j) = 0$, если i > 0 и y = O,

$P(i, j) = (P(i-1, j) + P(i,j-1))/2$, если i > Ои y > 0. (10.4)

На основании соотношений (10.4) можно показать, что вычисление $P(i, y)$ требует времени не больше, чем $O(2^{i+j})$. Обозначим через $T(n)$ максимальное время, которое требуется для вычисления $P(i, j)$, где i + y = n. Тогда из (10.4) следует

$T(1) = c$,

$T(n) = 2T(n - l) + d$,

для некоторых констант c и d. Читатель может проверить (воспользовавшись средствами, обсуждавшимися в предыдущей главе), что $T(n) < 2^{n-1}c + (2^{n-1} - 1)d$, что соответствует $O(2^n)$ или $O(2^{i+j})$.

Таким образом, мы установили экспоненциальную верхнюю границу времени, требуемого для рекурсивных вычислений $P(i, j)$. Но если мы хотим убедиться в том, что рекуррентная формула для $P(i, j)$ — не самый лучший способ вычисления этой вероятности, надо найти нижнюю границу времени вычисления. В качестве упражнения рекомендуем читателям показать, что при вычислении $P(i, j)$ общее количество вычислений вероятностей P с меньшими значениями i и y, которые приходится выполнять, равняется C_{i+j}^{i}, т.е. числу способов выбрать i элементов *из* $i + j$. Если $i - j$,

[1] Динамическим программированием (в наиболее общей форме) называют процесс пошагового решения задач, когда на каждом шаге выбирается одно решение из множества допустимых (на этом шаге) решений, причем такое, которое оптимизирует заданную целевую функцию или функцию критерия. (В приведенных далее примерах процесс пошагового решения задачи сводится к пошаговому заполнению таблиц.) В основе теории динамического программирования лежит *принцип оптимальности Беллмана*. Более подробно о теории динамического программирования см. [7]. — *Прим. ред.*

это число равно $\Omega(2^n/\sqrt{n})$, где $n = i + j$. Таким образом, $T(n)$ равняется $\Omega(2^n/\sqrt{n})$, и, по существу, мы можем показать, что оно также равняется $O(2^n/\sqrt{n})$. В то время как $2^n/\sqrt{n}$ растет медленнее, чем 2^n, разница все же невелика, и $T(n)$ растет настолько быстро, что использование рекуррентных вычислений $P(i, j)$ является совершенно неоправданным.

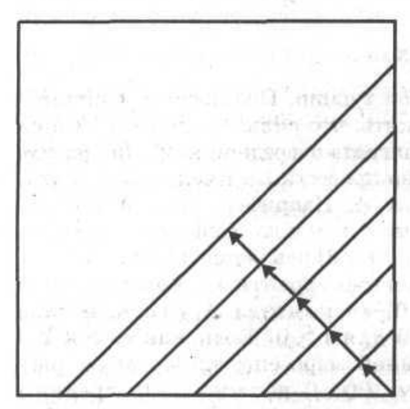


Рис. 10.4. Шаблон вычислений

Проблема с рекуррентными вычислениями заключается в том, что нам приходится периодически вычислять одно и то же значение $P(i, j)$ несколько раз. Если, например, надо вычислить $P(2, 3)$, то в соответствии с (10.4) требуется сначала вычислить $P(1, 3)$ и $P(2, 2)$. Однако для вычисления $P(1, 3)$ и $P(2, 2)$ нужно вычислить $P(1,2)$, т.е. одно и то же значение $P(1, 2)$ мы вычисляем дважды.

Более рациональным способом вычисления $P(i, j)$ является заполнение специальной таблицы (табл. 10.1), в нижней строке которой присутствуют одни нули, а крайний правый столбец целиком заполнен единицами (это соответствует первым двум строкам из (10.4)). Что же касается последней строки (10.4), то все остальные элементы таблицы представляют собой средние значения элементов, находящихся снизу и справа от соответствующего элемента. Таким образом, таблицу следует заполнять начиная с ее нижнего правого угла и продвигаться вверх и влево вдоль диагоналей, представляющих элементы с постоянным значением $i + j$, как показано на рис. 10.4. Соответствующая программа приведена в листинге 10.2 (предполагается, что она работает с двумерным массивом P подходящего размера).

Таблица 10.1. Таблица вероятностей

					j ↑
1/2	21/32	13/16	15/16	1	4
11/32	1/2	11/16	7/8	1	3
3/16	5/16	1/2	3/4	1	2
1/16	1/8	1/4	1/2	1	1
0	0	0	0		0
4	3	2	1	0	
		← i			

Листинг 10.2. Вычисление вероятностей

```
    function odds ( i, j: integer ): real;
        var
            s, k: integer;
        begin
(1)         for s:= 1 to i + j do begin
                { вычисление элементов массива P,
                    сумма индексов которых равняется s }
(2)             P[0,s]:= 1.0;
(3)             P[s,0]:= 0.0;
```

```
(4)             for k:= 1 to s - 1 do
(5)                 P[k, s-k]:= (P(k-l, s-k] + P[k, s-k-1])/2.0
            end;
(6)     return(P[i, j ] )
    end; { odds }
```

Анализ функции *odds* (шансы) не представляет особых затруднений. Цикл, включающий строки (4), *(5),* занимает O(s) времени, которое превосходит время $O(1)$ для строк (2), (3). Таким образом, выполнение внешнего цикла занимает время $O(\sum_{s=1}^{n} s)$ или $O(n^2)$, где $n = i + j$. Следовательно, вычисления по методу динамического программирования требуют времени $O(n^2)$, тогда как в "прямолинейном" подходе к вычислениям необходимо время порядка $O(2^n/\sqrt{n})$. Поскольку величина $2^n/\sqrt{n}$ растет намного быстрее, чем n^2, динамическое программирование предпочтительнее как в данном случае, так практически в любых других обстоятельствах.

Задача триангуляции

В качестве еще одного примера динамического программирования рассмотрим *задачу триангуляции многоугольника.* Заданы вершины многоугольника и расстояния между каждой парой вершин. Это расстояние может быть обычным евклидовым расстоянием на плоскости или произвольной функцией стоимости, задаваемой в виде таблицы. Задача заключается в том, чтобы выбрать такую совокупность *хорд* (линий между несмежными вершинами), что никакие две хорды не будут пересекаться, а весь многоугольник будет поделен на треугольники. Общая длина выбранных хорд должна быть минимальной. Такая совокупность хорд называется *минимальной триангуляцией.*

Пример 10.1. На рис. 10.5 показан многоугольник с семью сторонами, *(x, y)* — координаты его вершин. Расстояния вычисляются как обычное евклидово расстояние. Триангуляция, оказавшаяся не минимальной, изображена пунктирными линиями. Ее стоимостью является сумма длин хорд (v_0, v_2), (v_0, v_3), (v_0, v_5) и (v_3, v_5), или $\sqrt{8^2+16^2}+\sqrt{15^2+16^2}+\sqrt{22^2+2^2}+\sqrt{7^2+14^2} = 77.56$. П

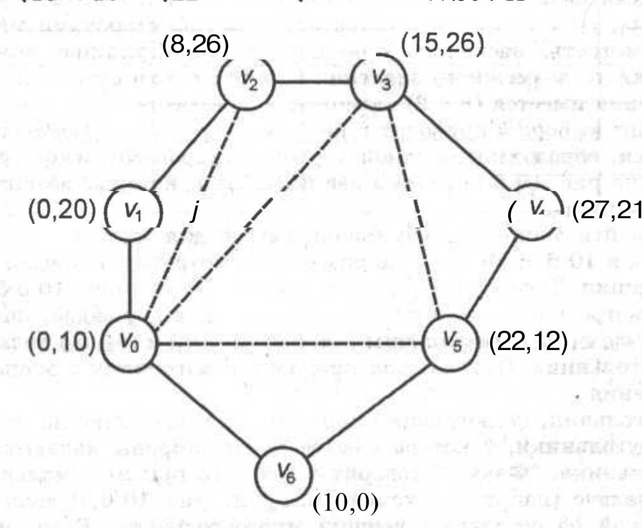


Рис. 10.5. Многоугольник и его триангуляция

Задача триангуляции, представляющая интерес сама по себе, имеет к тому же ряд полезных применений. Например, в статье [40] использовали обобщение задачи триангуляции для следующей цели. Рассмотрим задачу создания двумерной тени объекта, поверхность которого определяется совокупностью точек в трехмерном пространстве. Свет на объект падает из неподвижного источника, и яркость любой точки на поверхности зависит от углов между направлением света, положением наблюдателя и перпендикуляром (нормалью) к поверхности в этой точке. Чтобы определить нормаль к поверхности в какой-либо точке, надо найти минимальную триангуляцию точек, определяющих эту поверхность.

Каждый треугольник определяет некоторую плоскость в трехмерном пространстве, и, поскольку ищется минимальная триангуляция, можно предположить, что эти треугольники будут очень малы. Нетрудно определить направление нормали к плоскости, что позволяет вычислить интенсивность освещения для точек каждого треугольника. Если эти треугольники недостаточно малы, то, чтобы избежать резких перепадов в интенсивности освещения, общую картину можно улучшить путем локального усреднения.

Прежде чем продолжить решение задачи триангуляции с помощью динамического программирования, необходимо зафиксировать два факта относительно триангуляции. Они помогут разработать требуемый алгоритм. Предполагаем, что наш многоугольник имеет n вершин $v_0, v_1, \dots v_{n-1}$, перечисленных по часовой стрелке.

Факт 1. В случае триангуляции любого многоугольника, содержащего более трех вершин, с каждой парой смежных вершин связана по крайней мере одна хорда. Чтобы убедиться в этом, допустим, что ни v_i, ни v_{i+1}[1] не связаны ни с какой из хорд. В таком случае область, ограничиваемая стороной (v_i, v_{i+1}), должна была бы включать стороны $(v_{i-1}, v_i), (v_{i+1}, v_{i+2})$ и по крайней мере еще одну дополнительную сторону или хорду. В таком случае эта область не была бы треугольником.

Факт 2. Если (v_i, v_j) является хордой в триангуляции, значит, должна найтись такая вершина v_k, что каждая из линий (v_i, v_k) и (v_k, v_j) должна быть либо стороной многоугольника, либо хордой. В противном случае хорда (v_i, v_j)ограничивала бы область, не являющуюся треугольником.

Чтобы приступить к поиску минимальной триангуляции, выберем две смежные вершины, например v_0 и v_1. Из указанных выше фактов следует, что в любой триангуляции (и, следовательно, в минимальной триангуляции) должна найтись такая вершина v_k, что (v_1, v_k) и (v_k, v_0) являются хордами или сторонами многоугольника. Необходимо рассмотреть, насколько приемлемую триангуляцию можно построить после выбора каждого возможного значения k. Если в многоугольнике n вершин, в нашем распоряжении имеется (л — 2) возможных вариантов.

Каждый вариант выбора k приводит к не более чем двум *подзадачам,* где мы имеем многоугольники, образованные одной хордой и сторонами исходного многоугольника. Например, на рис. 10.6 показаны две подзадачи, которые возникают в случае, если выбрана вершина v_3.

Далее нужно найти минимальную триангуляцию для многоугольников, показанных на рис. 10.6,а и 10.6,б. Можно, например, рассмотреть все хорды, исходящие из двух смежных вершин. Тогда, решая задачу, показанную на рис. 10.6,б, мы должны, в частности, рассмотреть вариант хорды (v_3, v_5), которая порождает подзадачу с многоугольником (v_0, v_3, v_5, v_6), две стороны которого (v_0, v_3) и (v_3, v_5) являются хордами исходного многоугольника. Этот подход приводит к алгоритму с экспоненциальным временем выполнения..

Но, имея треугольник, включающий хорду (v_0, v_k), нам никогда не придется рассматривать многоугольники, у которых более одной стороны являются хордами исходного многоугольника. "Факт 2" говорит о том, что при минимальной триангуляции хорда в подзадаче (например, хорда (v_0, v_3) на рис. 10.6,б) должна составлять треугольник с одной из остальных вершин многоугольника. Если, например, сле-

[1] В дальнейшем предполагается, что все нижние индексы вычисляются по модулю n. Таким образом, v_i и v_{i+1} на рис. 10.5 могли бы быть v_6 и v_0 соответственно, поскольку п = 7.

дующей выбрать вершину v_4, то получим треугольник (v_0, v_3, v_4) и подзадачу (v_0, v_4, v_5, v_6), в которой осталась только одна хорда исходного многоугольника. Если мы выберем вершину v_5, то получим подзадачи (v_3, v_4, v_5) и (v_0, v_5, v_6) с хордами (v_3, v_5) и (v_0, v_5).

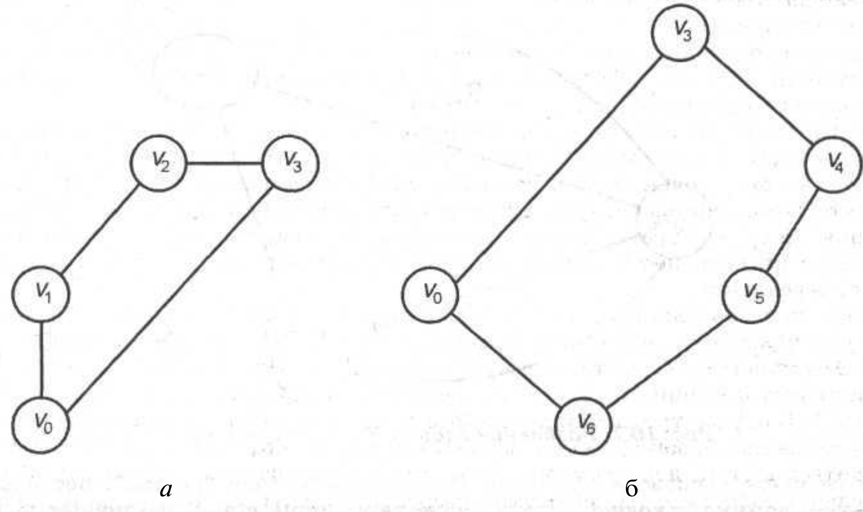


Рис. 10.6. Две подзадачи, возникающие после выбора вершины v_3

Определим *подзадачу размера* s, *начинающуюся с вершины* v_i и обозначаемую S_{is}, как задачу минимальной триангуляции для многоугольника, образованного s вершинами, начинающегося с вершины v_i и содержащего вершины $v_i, v_{i+1}, \dots, v_{i+s-1}$, перечисляемые в порядке обхода вершин многоугольника по часовой стрелке. В S_{is} хордой является замыкающая сторона (v_i, v_{i+s-1}). Например, многоугольник на рис. 10.6,а является подзадачей S_{04}, а на рис. 10.6,б — подзадачей S_{35}.[1] Чтобы решить подзадачу S_{is}, необходимо рассмотреть три следующих варианта.

1. Можно выбрать вершину v_{i+s-2}, чтобы составить треугольник с хордами (v_i, v_{i+s-1}) и (v_i, v_{i+s-2}) и третьей стороной (v_{i+s-2}, v_{i+s-1}), а затем решить подзадачу $S_{i,s-1}$.
2. Мы можем выбрать вершину v_{i+1}, чтобы составить треугольник с хордами (v_i, v_{i+s-1}) и (v_{i+1}, v_{i+s-1}) и третьей стороной (v_i, v_{i+1}), а затем решить подзадачу $S_{i+1,s-1}$.
3. Для некоторого k из диапазона от 2 до s - 3 можно выбрать вершину v_{i+k} и образовать треугольник со сторонами (v_i, v_{i+k}), (v_{i+k}, v_{i+s-1}) и (v_i, v_{i+s-1}), а затем решить подзадачи $S_{i,k+1}$ и $S_{i+k,s-k}$.

Если вспомним, что "решение" любой подзадачи размером не более трех не требует вообще никаких действий, то, рассматривая описанные варианты 1 – 3, следует предпочесть вариант 3, где надо выбрать некоторое k из диапазона от 1 до s - 2 и решить подзадачи $S_{i,k+1}$ и $S_{i+k,s-k}$. Рис. 10.7 иллюстрирует описанное разбиение на подзадачи.

Если для решения подзадач размером четыре и более воспользоваться очевидным рекурсивным алгоритмом, вытекающим из перечисленных выше вариантов, то можно показать, что каждое обращение к подзадаче размером s приводит к общему количеству рекурсивных вызовов, равному 3^{s-4}, если подсчитывать лишь обращения к

[1] Еще раз напомним, что номера вершин вычисляются по модулю га (n — количество вершин исходного многоугольника). Поэтому в подзадаче S_{35} вершиной v_{i+s-1} является вершина $v_{3+5-1} = v_7 = v_0$. — *Прим. ред.*

подзадачам размером четыре и более. Таким образом, количество подзадач, которые необходимо решить, возрастает по экспоненциальному закону в зависимости от s. Поскольку исходная задача имеет размер *n,* где п — количество вершин в заданном многоугольнике, общее количество шагов, выполняемых этой рекурсивной процедурой, возрастает экспоненциально по п.

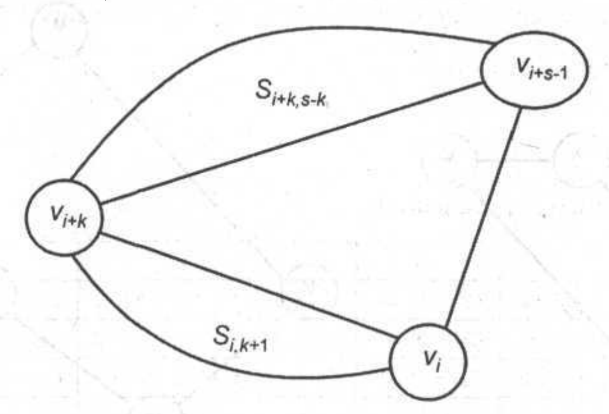


Рис. 10.7. Разбиение задачи S_{is} на подзадачи

Тем не менее в проведенном нами анализе что-то явно "не так", поскольку известно, что, помимо исходной задачи, существует лишь $n(n-4)$ различных подзадач, которые в любом случае необходимо решить. Их можно представить как подзадачи S_{is}, где $0 < i < n$ и $4 < s < л$. Очевидно, не все подзадачи, решенные с помощью рекурсивной процедуры, различаются между собой. Если, например, на рис. 10.5 выбрать хорду (v_0, v_3), а затем в подзадаче на рис. 10.6,б выбрать v_4, то придется решить подзадачу S_{44}. Но эту же задачу надо было бы решать, если бы сначала была выбрана хорда (v_0, v_4), а затем, решая подзадачу S_{45}, выбрана вершина v_0 (чтобы замкнуть треугольник с вершинами v_1 и v_4).

Это подсказывает нам эффективный способ решения задачи триангуляции. Мы составляем таблицу, назначая стоимость C_{is} триангуляции S_{is} для всех i и s. Поскольку решение любой данной задачи зависит лишь от решения задач меньшего размера, логическим порядком заполнения такой таблицы является порядок, основанный на размерах подзадач, т.е. для размеров s = 4, 5,- ..., *n* - 1 мы указываем минимальную стоимость задач S_{is} для всех вершин г. Удобно включить и задачи размеров $0 < s < 4$, но при этом нужно помнить, что S_{is} имеет стоимость 0, если $s < 4$.

В соответствии с перечисленными выше вариантами действий 1 – 3 при определении подзадач формула вычисления C_{is} при $s > 4$ должна иметь следующий вид:

$$C_{is} = \min_{1 \le k \le s-2} \left[C_{i,k+1} + C_{i+k,s-k} + D(v_i, v_{i+k}) + D(v_{i+k}, v_{i+s-1}) \right], \quad (10.5)$$

где $D(v_p,v_q)$ — длина хорды между вершинами v_p и v_q, если v_p и v_q не являются смежными вершинами многоугольника; $D(v_p,v_q)$ равняется 0, если v_p и v_q являются смежными вершинами.

Пример 10.2. В табл. 10.2 приведены затраты для триангуляции $S_{i,s}$ при $0 < i < 6$ и $4 < s < 6$, причем за основу взят многоугольник и расстояния, показанные на рис. 10.5. Все затраты для строк с $s<3$ равны нулю. Мы заполнили элемент C_{07} (столбец 0 и строка для s = 7). Этот элемент, как и все остальные элементы этой строки, представляет стоимость триангуляции всего многоугольника. Чтобы убедиться в этом, нужно лишь обратить внимание на то, что мы можем рассматривать сторону (v_0, v_6) как хорду большего многоугольника, а многоугольник, показанный на рис. 10.5, — как подзадачу этого многоугольника, которая включает ряд до-

полнительных вершин, располагающихся по часовой стрелке от вершины v_6 до вершины v_0. Обратите внимание, что вся строка для s = 7 должна иметь то же значение, что и C_{07}, по крайней мере, в пределах погрешности вычислений.

В качестве примера покажем, как заполняется элемент в столбце для $i = 6$ и в строке для s = 5. В соответствии с (10.5) значение этого элемента (C_{65}) представляет собой минимальное значение из трех сумм, соответствующих $k = 1$, 2 или 3. Вот эти суммы:

$$C_{62} + C_{04} + D(v_6, v_0) + D(v_0, v_3),$$
$$C_{63} + C_{13} + D(v_6, v_1) + D(v_1, v_3),$$
$$C_{64} + C_{22} + D(v_6, v_2) + D(v_2, v_3).$$

Таблица 10.2. Таблица стоимостей C_{is}

7	C_{07} = 75.43						
6	C_{06} = 53.34	C_{16} = 55.22	C_{26} = 57.54	C_{36} = 59.67	C_{46} = 59.78	C_{56} = 59.78	C_{66} = 63.61
5	C_{05} = 37.54	C_{15} = 31.81	C_{25} = 35.45	C_{35} = 37.74	C_{45} = 45.50	C_{55} = 39.98	C_{65} = 38.09
4	C_{04} = 16.16	C_{14} = 16.16	C_{24} = 15.65	C_{34} = 15.65	C_{44} = 22.09	C_{54} = 22.09	C_{64} = 17.89
s	$i = 0$	1	2	3	4	5	6

Требуемые нам расстояния вычисляются на основе координат вершин следующим образом: $D(v_2, v_3) = D(v_6, v_0) = 0$ (поскольку это стороны многоугольника, а не хорды, и предоставляются "бесплатно"), далее:

$$D(v_6, v_2) = 26.08,$$
$$D(v_1, v_3) = 16.16,$$
$$D(v_6, v_1) = 22.36,$$
$$D(v_0, v_3) = 21.93.$$

Указанными тремя суммами являются 38.09, 38.52 и 43.97 соответственно. Отсюда видно, что минимальная стоимость подзадачи S_{65} равняется 38.09. Более того, поскольку первая сумма оказалась наименьшей, то теперь мы знаем, что для достижения этого минимума надо использовать подзадачи S_{62} и S_{04}, т.е. выбрать хорду (v_0, v_3), а затем найти оптимальное решение для S_{64}; для этой подзадачи хорда (v_1, v_3) является предпочтительным вариантом. □

Заполнять табл. 10.2 удобно приемом, который вытекает из формулы (10.5). Каждое выражение в (10.5) требует пары элементов. В первую пару для $k = 1$ входит элемент из самого низа таблицы (строка для s = 2)[1] в столбце вычисляемого элемента и элемент, который находится на одну строку ниже и справа от вычисляемого элемента[2]. Первый элемент второй пары находится на один ряд выше "дна" таблицы в столбце вычисляемого элемента, второй отстоит от вычисляемого элемента на две позиции вниз и вправо. На рис. 10.8 показаны две линии элементов, следуя за которыми получим все пары элементов, которые будут анализироваться одновременно. Модель продвижения — вверх по столбцу и вниз по диагонали — является общепринятой при заполнении таблиц в методе динамического программирования.

---

[1] Не забывайте, что табл. 10.2 содержит нулевые строки ниже тех, которые показаны.

[2] Говоря "справа", мы подразумеваем таблицу с циклическим возвратом из конца в начало. Таким образом, если мы находимся в крайнем справа столбце, то столбцом "справа" от него будет крайний слева столбец.

Поиск решений на основе таблицы

Несмотря на то что табл. 10.2 позволяет определить стоимость минимальной триангуляции, из нее непосредственно нельзя определить саму минимальную триангуляция, так как для этого надо знать значение k, которое обеспечивает минимум в формуле (10.5). Если мы знаем значение k, тогда решение состоит из хорд (v_i, v_{i+k}) и (v_{i+k}, v_{i+s-1}) (за исключением случая, когда одна из них не является хордой, так как $k = 1$ или $k = s - 2$) плюс хорды, указываемые решениями $S_{i,k+1}$ и $S_{i+k,s-k}$. При вычислении элементов таблицы полезно включить в нее значения k, при которых получаем минимум в формуле (10.5).

Пример 10.3. В табл. 10.2 значение элемента C_{07} (который представляет решение задачи триангуляции многоугольника из рис. 10.5) получено как минимум формулы (10.5) при $k = 5$. Другими словами, задача S_{07} разбивается на подзадачи S_{06} и S_{52}: первая из них представляет собой задачу с шестью вершинами $v_0, v_1, \ldots, v_5$, а вторая является тривиальной "задачей" стоимости О. Таким образом, мы имеем хорду (v_0, v_5) стоимостью 22.09 и должны решить подзадачу S_{06}.

Минимальная стоимость для C_{06} получается из (10.5) при $k = 2$, в результате чего задача S_{06} разбивается на подзадачи S_{03} и S_{24}. Первая из них представляет собой треугольник с вершинами v_0, v_1 и v_2, в то время как вторая представляет собой четырехугольник, определяемый вершинами v_2, v_3, v_4 и v_5. S_{03} решать не требуется, нужно решить только подзадачу S_{24}. Кроме того, надо включить в стоимость минимальной триангуляции стоимости хорд (v_0, v_2) и (v_2, v_5), равные соответственно 17.89 и 19.80. Для C_{24} минимальное значение в (10.5) получается при $k = 1$, давая подзадачи C_{22} и Сзз, причем обе они имеют размер, не больший трех, и, следовательно, их стоимость равна 0. Вводим хорду (v_3, v_5) стоимостью 15.65 в минимальную триангуляцию. □

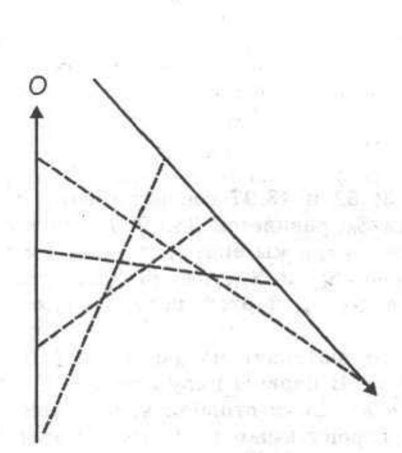


Рис. 10.8. *Шаблон просмотра таблицы при вычислении одного элемента*

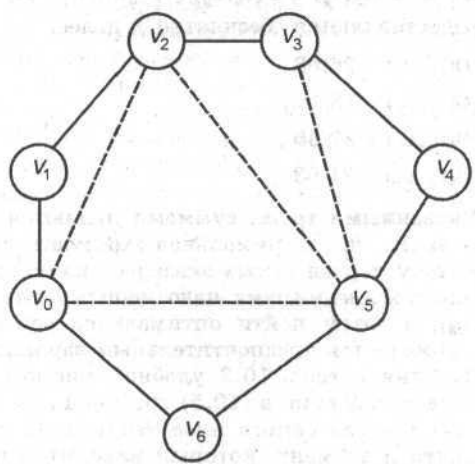


Рис. 10.9. *Триангуляция с минимальной стоимостью*

10.3. „Жадные" алгоритмы

Рассмотрим небольшую "детскую" задачу. Допустим, что у нас есть монеты достоинством 25, 10, 5 копеек и 1 копейка и нужно вернуть сдачу 63 копейки. Почти не раздумывая, мы преобразуем эту величину в две монеты по 25 копеек, одну монету в 10 копеек и три монеты по одной копейке. Нам не только удалось быстро определить перечень монет нужного достоинства, но и, по сути, мы составили самый короткий список монет требуемого достоинства.

Алгоритм, которым читатель в этом случае наверняка воспользовался, заключался в выборе монеты самого большого достоинства (25 копеек), но не больше 63 копеек, добавлению ее в список сдачи и вычитанию ее стоимости из 63 (получается 38 копеек). Затем снова выбираем монету самого большого достоинства, но не больше остатка (38 копеек): этой монетой опять оказывается монета в 25 копеек. Эту монету мы опять добавляем в список сдачи, вычитаем ее стоимость из остатка и т.д.

Этот метод внесения изменений называется *"жадным" алгоритмом*. На каждой отдельной стадии "жадный" алгоритм выбирает тот вариант, который является ***локально оптимальным*** в том или ином смысле. Обратите внимание, что алгоритм для определения сдачи обеспечивает в целом оптимальное решение лишь вследствие особых свойств монет. Если бы у нас были монеты достоинством 1 копейка, 5 и 11 копеек и нужно было бы дать сдачу 15 копеек, то "жадный" алгоритм выбрал бы сначала монету достоинством 11 копеек, а затем четыре монеты по одной копейке, т.е. всего пять монет. Однако в данном случае можно было бы обойтись тремя монетами по 5 копеек.

Мы уже встречались в этой книге с несколькими "жадными" алгоритмами, например алгоритмом построения кратчайшего пути Дейкстры и алгоритмом построения остовного дерева минимальной стоимостью Крускала. Алгоритм кратчайшего пути Дейкстры является "жадным" в том смысле, что он всегда выбирает вершину, ближайшую к источнику, среди тех, кратчайший путь которых еще неизвестен. Алгоритм Крускала также "жадный"; он выбирает из остающихся ребер, которые не создают цикл, ребро с минимальной стоимостью.

Следует подчеркнуть, что не каждый "жадный" алгоритм позволяет получить оптимальный результат в целом. Как нередко бывает в жизни, "жадная стратегия" подчас обеспечивает лишь сиюминутную выгоду, в то время как в целом результат может оказаться неблагоприятным. Посмотрим, например, что произойдет, если в алгоритмах Дейкстры и Крускала допустить наличие ребер с отрицательными весами. Оказывается, что на алгоритм Крускала построения остовного дерева это никак не повлияет: с его помощью по-прежнему можно будет получить дерево минимальной стоимости. Но алгоритм Дейкстры в некоторых случаях уже не позволяет получить кратчайшие пути.

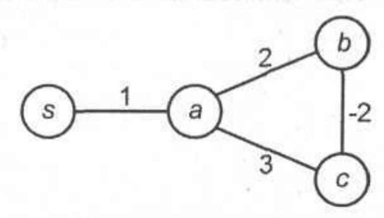


Рис. 10.10. Граф с ребром отрицательного веса

Пример 10.4. На рис. 10.10 показан граф с ребром отрицательной стоимости между вершинами b и c. Если источником является вершина s, то алгоритм Дейкстры сначала правильно определяет, что минимальный путь до a имеет протяженность 1. Теперь, рассматривая ребра от s (или a) до b или c, алгоритм рассчитывает, что кратчайший путь от s до b, а именно $s \to a \to b$, имеет длину 3. Но далее получаем, что c имеет кратчайший путь от s длиной 1.

Однако "жадный" выбор b вместо c является неоправданным. Оказывается, что путь $s \to a \to c \to b$ имеет длину лишь 2, поэтому вычисленное нами минимальное расстояние для b является неправильным.[1] □

„Жадные" алгоритмы как эвристики

Существуют задачи, для которых ни один из известных "жадных" алгоритмов не позволяет получить оптимального решения; тем не менее имеются "жадные" алгоритмы, которые с большой вероятностью позволяют получать "хорошие" решения. Нередко вполне удовлетворительным можно считать "почти оптимальное" решение,

[1] Вообще говоря, когда допускается наличие ребер с отрицательными весами (стоимостью), к понятию "кратчайший путь" нужно относиться очень осторожно. Если, например, допускается наличие циклов с отрицательной стоимостью, то ничто не мешает многократно проходить такой цикл, получая при этом сколь угодно большие отрицательные расстояния. Поэтому было бы вполне естественным ограничиться лишь ациклическими путями.

характеризующееся стоимостью, которая лишь на несколько процентов превышает оптимальную. В таких случаях "жадный" алгоритм зачастую оказывается самым быстрым способом получить "хорошее" решение. Вообще говоря, если рассматриваемая задача такова, что единственным способом получить оптимальное решение является использование метода полного поиска, тогда "жадный" алгоритм или другой эвристический метод получения хорошего (хотя и необязательно оптимального) решения может оказаться единственным реальным средством достижения результата.

Пример 10.5. Рассмотрим одну знаменитую задачу, для получения оптимального решения которой из всех известных нам алгоритмов подходят лишь алгоритмы полного перебора, время выполнения которых зависит экспоненциально от объема входных данных. Эта задача, которая называется *задачей коммивояжера,* сводится к поиску в неориентированном графе с весовыми значениями ребер такого *маршрута* (простого цикла, включающего все вершины), у которого сумма весов составляющих его ребер будет минимальной. Такой маршрут часто называют ***гамильтоновым*** *циклом*.

На рис. 10.11,а показан граф с шестью вершинами (их часто называют "городами"), который может служить основой для задачи коммивояжера. Заданы координаты каждой вершины, а весом каждого ребра считается его длина. Обратите внимание: мы предполагаем (и такое предположение характерно для задачи коммивояжера), что существуют все ребра графа, т.е. граф является полным. В более общих случаях, когда вес ребер не основывается на евклидовом расстоянии, у ребра может оказаться бесконечный вес, чего в действительности, конечно, не бывает.

На рис. 10.11, б, в, г, д показаны четыре маршрута по шести "городам". Читатель может самостоятельно прикинуть, какой из этих маршрутов является оптимальным (может быть, такого маршрута вообще нет). Протяженности этих четырех маршрутов равняются 50.00, 49.73, 48.39 и 49.78 соответственно; маршрут на рис. 10.11,гявляется кратчайшим из всех возможных маршрутов.

c • (1,7) d • (15,7)
e • (15,4)
b • (4,3)
a • (0,0) f • (18,0)

а. Шесть "городов"

б в

г д

Рис. 10.11. Пример задачи коммивояжера

Задача коммивояжера имеет ряд практических применений. Как следует из самого названия задачи, ее можно использовать для составления маршрута человека, который должен посетить ряд пунктов и, в конце концов, вернуться в исходный пункт. Например, задача коммивояжера использовалась для составления маршрутов лиц, занимающихся выемкой монет из таксофонов. В этом случае вершинами являются места установки таксофонов и "базовый пункт". Стоимостью каждого ребра (отрезка маршрута) является время в пути между двумя точками (вершинами) на маршруте.

Еще одним применением задачи коммивояжера является *задача обхода доски шахматным конем:* надо найти последовательность ходов, которые позволят коню обойти все поля шахматной доски, попадая на каждое поле лишь один раз, и вернуться, в конце концов, на исходное поле. В этом случае роль вершин графа выполняют поля шахматной доски. Предполагается также, что ребра между двумя полями, которые являются "составными частями" хода коня, имеют нулевой вес; все остальные ребра имеют вес, равный бесконечности. Оптимальный маршрут имеет нулевой вес и должен быть маршрутом коня. Читатель, возможно, удивится, узнав, что поиск маршрутов коня с помощью хороших эвристических алгоритмов для задачи коммивояжера вообще не составляет проблемы, в то время как поиск "вручную" является весьма непростой задачей.

"Жадный" алгоритм для задачи коммивояжера, речь о котором пойдет ниже, является вариантом алгоритма Крускала. Здесь, как и в основном алгоритме Крускала, сначала рассматриваются самые короткие ребра. В алгоритме Крускала очередное ребро принимается в том случае, если оно не образует цикл с уже принятыми ребрами; в противном случае ребро отвергается. В случае задачи коммивояжера "критерием принятия" ребра является то, что рассматриваемое ребро (в сочетании с уже принятыми ребрами)

- не приводит к появлению вершины со степенью три и более[1];
- не образует цикл (за исключением случая, когда количество выбранных ребер равняется количеству вершин в рассматриваемой задаче).

Совокупности ребер, выбранных в соответствии с этими критериями, образуют совокупность несоединенных путей; такое положение сохраняется до выполнения последнего шага, когда единственный оставшийся путь замыкается, образуя маршрут.

На рис. 10.11,а сначала выбирается ребро *(d, e),* поскольку оно является кратчайшим, имея длину 3. Затем рассматриваются ребра *(b, c)*, *(a, b)* и *(e, f)* — длина всех этих ребер равна 5. В каком порядке они будут рассмотрены, значения не имеет — все они удовлетворяют условиям для выбора, и мы должны выбрать их, если собираемся строго следовать "жадному методу". Следующим кратчайшим ребром является ребро *(a, c),* длина которого равна 7.08. Однако это ребро может образовать цикл с ребрами *(a, b)* и *(b, c),* поэтому принимаем решение отвергнуть его. Затем по той же причине отвергаем ребро *(d, f)*. Далее надо рассмотреть ребро *(b, e),* но его также придется отвергнуть, поскольку оно повышает степени вершин *b* и *e* до трех и не образует маршрут с теми ребрами, которые уже отобраны. Точно так же отвергается ребро *(b, d).* Затем рассматриваем ребро *(c, a)* и принимаем его. Итак, уже образовался путь $a \to b \to c \to d \to e \to f$, и, завершая маршрут, выбираем ребро *(a, f).* В результате получаем маршрут, показанный на рис. 10.11,б. Он по своей "оптимальности" находится на четвертом месте среди всех возможных маршрутов; его стоимость всего лишь на 4% больше стоимости оптимального маршрута.

10.4. Поиск с возвратом

Иногда приходится иметь дело с задачей поиска оптимального решения, когда невозможно применить ни один из известных методов, способных помочь отыскать оптимальный вариант решения, и остается прибегнуть к последнему средству — полному перебору. Этот раздел посвящен систематическому описанию метода полного перебора, называемого поиском с возвратом, а также метода, называемого альфа-бета отсечением, который зачастую позволяет существенно сократить объем операций поиска.

Рассмотрим какую-либо игру с участием двух игроков, например шахматы, шашки или "крестики-нолики". Игроки попеременно делают ходы, и состояние игры отражается соответствующим положением на доске. Допустим, что есть конечное число позиций на доске и в игре предусмотрено определенное "правило остановки", являющееся критерием ее завершения. С каждой такой игрой можно ассоциировать дерево, называемое *деревом*

[1] Степенью вершины называется количество ребер, инцидентных этой вершине. — *Прим. ред.*

игры. Каждый узел такого дерева представляет определенную позицию на доске. Начальная позиция соответствует корню дерева. Если позиция *x* ассоциируется с узлом *n*, тогда потомки узла *n* соответствуют совокупности допустимых ходов из позиции *x*, и с каждым потомком ассоциируется соответствующая результирующая позиция на доске. Например, на рис. 10.12 показана часть дерева для игры в "крестики-нолики".

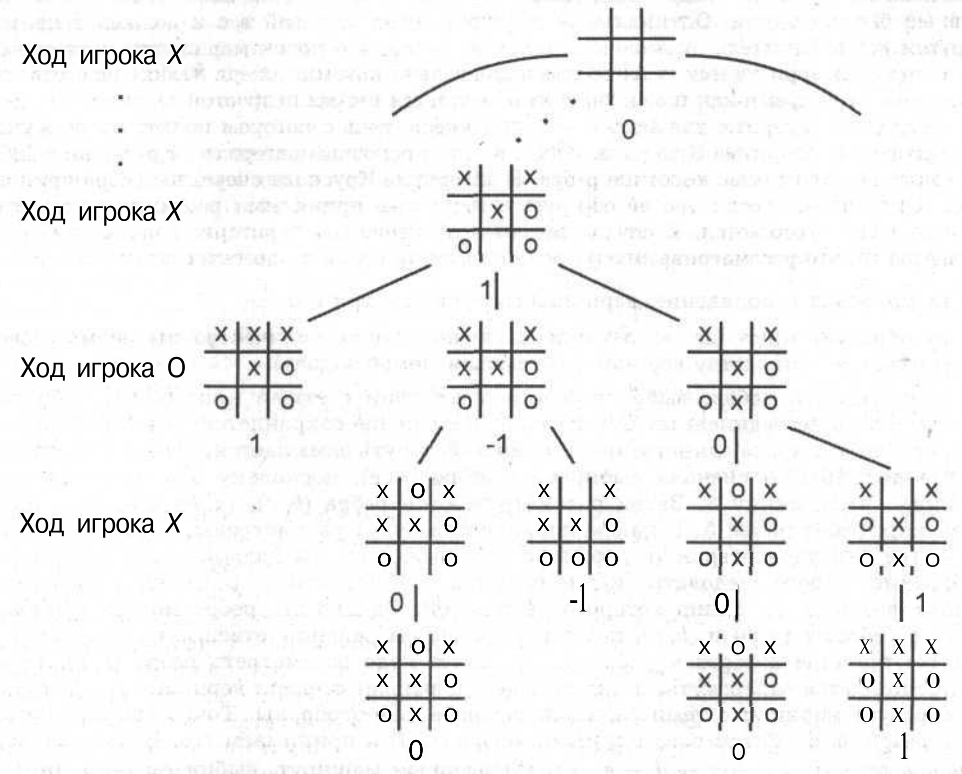


Рис. 10.12. Часть дерева игры в "крестики–нолики"

Листья этого дерева соответствуют таким позициям на доске, из которых невозможно сделать ход, — то ли потому, что кто-то из игроков уже одержал победу, то ли потому, что все клетки заполнены и игра закончилась вничью. Каждому узлу дерева соответствует определенная цена. Сначала назначается цены листьям. Допустим, речь идет об игре в "крестики–нолики". В таком случае листу назначается цена -1, 0 или 1 в зависимости от того, соответствует ли данной позиции проигрыш, ничья или выигрыш игрока 1 (который ставит "крестики").

Эти цены распространяются вверх по дереву в соответствии со следующим правилом. Если какой-либо узел соответствует такой позиции, из которой должен сделать ход игрок 1, тогда соответствующая цена является максимальной среди цен потомков данного узла. При этом предполагаем, что игрок 1 сделает самый выгодный для себя ход, т.е. такой, который принесет ему самый ценный результат. Если же узел соответствует ходу игрока 2, тогда соответствующая цена является минимальной среди цен потомков данного узла. При этом мы предполагаем, что игрок 2 также сделает самый выгодный для себя ход, т.е. такой, который при самом благоприятном исходе приведет к проигрышу игрока 1, а при менее предпочтительном исходе — к ничьей.

Пример 10.6. На рис. 10.12 показаны цены позиций в игре "крестики-нолики". Листьям, которые приносят выигрыш для игрока *O* (который рисует "нолики"), назначается цена -1; листьям, которые приносят ничью, назначается цена О; а листь-

ям, которые приносят выигрыш для игрока *X* (который рисует "крестики"), назначается цена +1. Затем продвигаемся вверх по дереву. На уровне 8, где остается только одна пустая клетка, и предстоит ход игроку *X*, ценой для неразрешенных позиций является "максимум" из одного потомка на уровне 9.

На уровне 7, где предстоит ход игроку *O*, и имеются два варианта решения, в качестве цены для внутреннего узла принимаем минимальную из цен его потомков. Крайняя слева позиция на уровне 7 представляет собой лист и имеет цену 1, поскольку она является выигрышем для *X*. Вторая позиция на уровне 7 имеет цену min(0, -1) = -1, тогда как третья имеет цену min(0, 1) = 0. Одна позиция, показанная на уровне 6 (на этом уровне предстоит ход игроку *X*), имеет цену max(1, -1, 0) = 1. Это означает, что для обеспечения выигрыша игроку *X* нужно сделать определенный выбор и в этом случае выигрыш последует немедленно. □

Обратите внимание: если цена корня равняется 1, то выигрышная стратегия находится в руках игрока 1. Действительно, когда наступает его очередь ходить, он гарантирован, что может выбрать ход, который приведет к позиции, имеющей цену 1. Когда же наступает очередь игрока 2 делать свой ход, ему не остается ничего иного, как выбрать ход, ведущий к все той же позиции с ценой 1, что для него означает проигрыш. Тот факт, что игра считается завершившейся, гарантирует в конечном счете победу первого игрока. Если же цена корня равняется О, как в случае игры в "крестики-нолики", тогда ни один из игроков не располагает выигрышной стратегией и может гарантировать себе лишь ничью, если будет играть без ошибок. Если же цена корня равняется -1, то выигрышная стратегия находится в руках игрока 2.

Функции выигрыша

Идею дерева игры, узлы которого имеют цену -1, 0 или 1, можно обобщить на деревья, листьям которых присваиваются любые числа (такое число называется *выигрышем),* выполняющие роль цены игры. Для оценивания внутренних узлов применяются те же правила: взять максимум среди потомков на тех уровнях, где предстоит сделать ход игроку 1, и минимум среди потомков на тех уровнях, где предстоит сделать ход игроку 2.

В качестве примера, где удобно применить концепцию выигрышей, рассмотрим сложную игру (например, шахматы), в которой дерево игры, являясь, в принципе, конечным, столь огромно, что любые попытки оценить его по методу "снизу вверх" обречены на неудачу. Любые шахматные программы действуют, в сущности, путем построения для каждой промежуточной позиции на шахматной доске дерева игры. Корнем этого дерева является текущая позиция; корень распространяется вниз на несколько уровней. Точное количество уровней зависит от быстродействия конкретного компьютера. Когда большинство листьев дерева проявляют "неоднозначность" (т.е. не ведут ни к выигрышу, ни к ничьей, ни к проигрышу), каждая программа использует определенную функцию позиций на доске, которая пытается оценить вероятность выигрыша компьютера в этой позиции. Например, на такой оценке в значительной мере сказывается наличие у одной из сторон материального перевеса и такие факторы, как прочность защиты королей. Используя подобную функцию выигрыша, компьютер может оценить вероятность выигрыша после выполнения им каждого из возможных очередных ходов (в предположении, что игра каждой из сторон будет проходить по оптимальному сценарию) и выбрать ход с наивысшим выигрышем.[1]

---

[1] Ниже перечислен ряд других правил, в соответствии с которыми действуют хорошие шахматные программы:

1. Использование эвристик, чтобы исключить из рассмотрения определенные ходы, которые вряд ли можно считать хорошими. Это помогает увеличить количество просматриваемых уровней дерева за фиксированное время.
2. Распространение "цепочек взятия", которые представляют собой последовательности ходов, сопровождающиеся взятием фигур противника, за пределы последнего уровня, до которого обычно распространяется дерево. Это помогает более точно оценить относительную материальную силу позиций.
3. Сокращение поиска на дереве методом альфа-бета отсечения (см. дальше в этом разделе).

Реализация поиска с возвратом

Допустим, что заданы правила некоторой **игры,**[1] т.е. допустимые ходы и правила завершения игры. Мы хотим построить дерево этой **игры** и оценить его корень. Это дерево можно построить обычным способом, а затем совершить обход его узлов в обратном порядке. Такой обход в обратном порядке гарантирует, что мы попадем на внутренний узел *n* только после обхода всех его потомков, в результате чего можно оценить узел л, найдя максимум или минимум (в зависимости от конкретной ситуации) значений всех его потомков.

Объем памяти, который требуется для хранения такого дерева, может оказаться недопустимо большим, но, если соблюдать определенные меры предосторожности, можно обойтись хранением в памяти в любой заданный момент времени лишь одного пути — от корня к тому или иному узлу. В листинге 10.3 показана рекурсивная программа *search* (поиск), которая выполняет обход дерева с помощью последовательности рекурсивных вызовов этой процедуры. Эта программа предполагает выполнение следующих условий.

1. Выигрыши являются действительными числами из конечного интервала, например от -1 до +1.
2. Константа ∞ больше, чем любой положительный выигрыш, а −∞ — меньше, чем любой отрицательный выигрыш.
3. Тип данных modetype (тип режима) определяется следующим образом:

```
type
    modetype = (MIN, MAX)
```

4. Предусмотрен тип данных **board**type (тип игровой доски), который определяется способом, подходящим для представления позиций на игровой доске.
5. Предусмотрена функция *payoff* (выигрыш), которая вычисляет выигрыш для любой позиции, которая является листом.

Листинг **10.3.** Рекурсивная программа **поиска с возвратом**

```
    function search ( B: boardtype; mode: modetype): real;
        { оценивает и возвращает выигрыш для позиции B в предположении,
            что следующим должен ходить игрок 1 (mode = MAX)
            или игрок 2 (mode = MIN) }
        var
            C: boardtype; { сын позиции B }
            value: real; { для временного хранения минимального или
                                максимального значения }
        begin
(1)         if B является листом then
(2)             return(payoff(B))
            else begin
(3)             if mode = MAX then
(4)                 value:= -∞
                else
(5)                 value:= ∞;
(6)             for для каждого сына C позиции B do
(7)                 if mode = MAX then
(8)                     value:= max(value, search(C, MIN))
                    else
```

[1] Не следует считать, что таким способом отыскиваются решения только для игр. Как будет показано в последующих примерах, "игра" может на самом деле представлять решение той или иной практической задачи.

```
(9)                value:= min(value, search(C, MAX));
(10)           return(value)
        end
    end; { search }
```

Можно рассмотреть еще один вариант реализации поиска с возвратом. В этом варианте используется нерекурсивная **программа,** которая поддерживает стек позиций, соответствующих последовательности вызовов функции *search.* При создании такой программы можно использовать методы, рассмотренные в разделе 2.6.

Альфа-бета отсечение

Одно простое условие позволяет нам избавиться от рассмотрения значительной части типичного дерева игры. Вернемся к эскизу программы, представленному в листинге 10.3. Цикл for в строке (6) может проигнорировать определенных сыновей — нередко довольно большое их число. Допустим, мы рассматриваем узел *n,* показанный на рис. 10.13, и уже определили, что цена c_1 первого из сыновей узла *n* равняется 20. Поскольку узел *n* находится в режиме MAX (т.е. ход 1-го игрока), то его значение не меньше 20. Допустим теперь, что, продолжая поиск, мы нашли, что у сына c_2 есть потомок *d* с выигрышем 15. Поскольку узел c_2 находится в режиме MIN (т.е. ход 2-го игрока), то значение узла c_2 не может быть больше 15. Таким образом, какое бы ни было значение узла c_2, оно не может влиять на цену узла *n* и любого предка узла га.

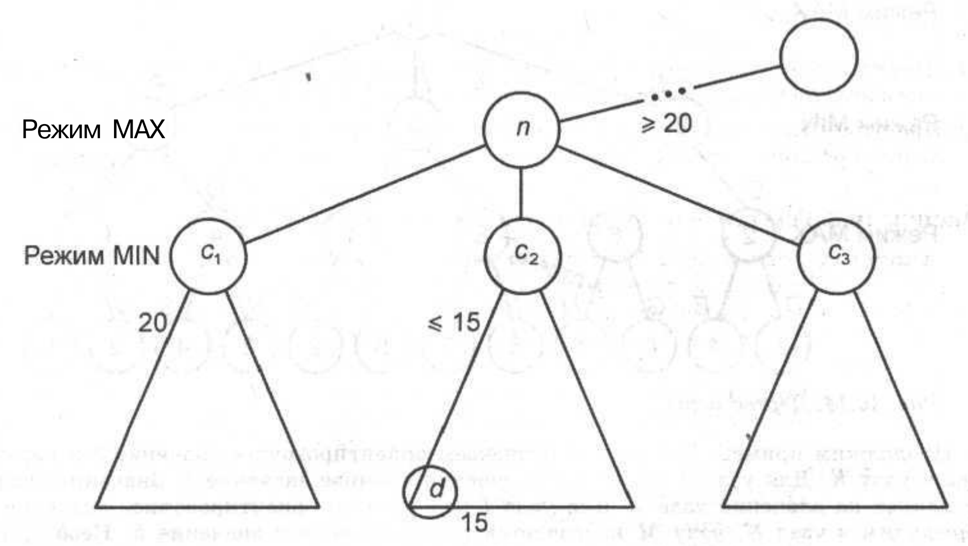


Рис. 10.13. Отсечение потомков узла

Таким образом, в ситуации, показанной на рис. 10.13, можно проигнорировать рассмотрение тех потомков узла c_2, которые мы еще не рассматривали. Общее правило пропуска или "отсечения" узлов связано с понятием конечных и ориентировочных значений для узлов. *Конечное значение* — это то, что мы просто называем "значением" (выигрышем). *Ориентировочное значение* — это верхний предел значений узлов в режиме MIN или нижний предел значений узлов в режиме MAX. Ниже перечислены правила вычисления конечных и ориентировочных значений.

1. Если мы уже рассмотрели или отсекли всех потомков узла га, сделать ориентировочное значение узла га конечным значением.
2. Если ориентировочное значение узла га в режиме MAX равно v_1, а конечное значение одного из его потомков равняется v_2, тогда установить ориентировочное

значение узла n равным $\max(v_1, v_2)$. Если же узел n находится в режиме MIN, тогда ориентировочное значение узла n установить равным $\min(v_1, v_2)$.

3. Если p является узлом в режиме MIN, имеет родителя q (в режиме MAX), а ориентировочные значения узлов p и q равняются v_1 и v_2 соответственно, причем $v_1 < v_2$, тогда можно отсечь всех нерассмотренных потомков узла p. Можно также отсечь нерассмотренных потомков узла p, если p является узлом в режиме MAX, а q является, таким образом, узлом в режиме MIN, и $v_2 < v_1$.

Пример 10.7. Рассмотрим дерево, показанное на рис. 10.14. Полагая, что значения листьев соответствуют значениям, указанным на рисунке, мы хотим вычислить значение корня. Начинаем обход дерева в обратном порядке. Достигнув узла *D,* в соответствии с правилом (2) назначаем узлу *C* ориентировочное значение 2, которое является конечным значение узла *D.* Просматриваем узел *E* и возвращаемся в узел С, а затем переходим в узел *B.* В соответствии с правилом (1) конечное значение узла *C* равно 2, а ориентировочное значение узла *B* — 2. Обход далее продолжается вниз к узлу *G, а* затем обратно в узел *F.* Ориентировочное значение узла *F* равно 6. В соответствии с правилом (3) можно отсечь узел *H,* поскольку ориентировочное значение узла *F* уменьшиться не может и оно уже больше значения узла *B,* которое не может увеличиться.

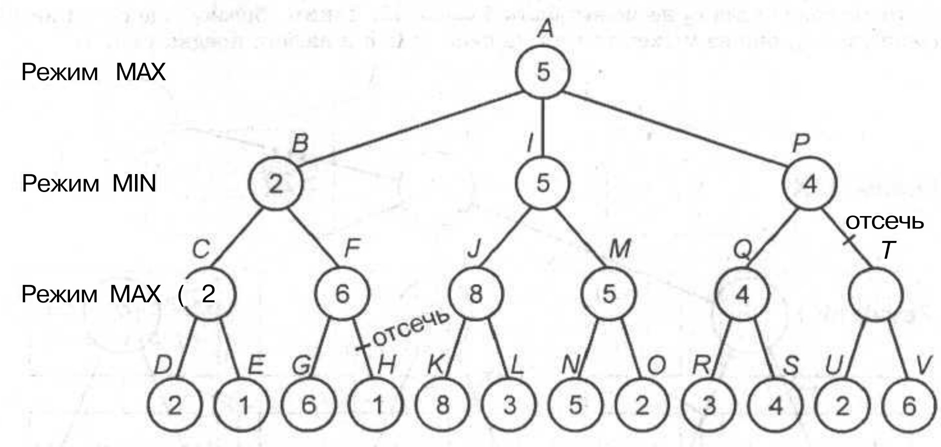


Рис. 10.14. Дерево игры

Продолжим пример. Для узла *A* назначаем ориентировочное значение 2 и переходим к узлу *K*. Для узла *J* назначается ориентировочное значение 8. Значение узла *L* не влияет на значение узла *J*. Для узла *I* назначается ориентировочное значение 8. Переходим в узел *N*, узлу *M* назначается ориентировочное значение 5. Необходимо выполнить просмотр узла *O,* поскольку 5 (ориентировочное значение узла *M)* меньше, чем ориентировочное значение узла *I*. Далее пересматриваются ориентировочные значения узлов *I* и А. Переходим к узлу *R.* Выполняется просмотр узлов *R* и S, узлу *P* назначается ориентировочное значение 4. Просмотр узла *T* и всех его потомков проводить не нужно, поскольку это может только понизить значение узла *P,* которое уже и без того слишком низкое, чтобы повлиять на значение узла А. П

Метод ветвей и границ

Игры — не единственная категория задач, которые можно решать полным перебором всего дерева возможностей. Широкий спектр задач, в которых требуется найти минимальную или максимальную конфигурацию того или иного типа, поддаются решению путем поиска с возвратом, применяемого к дереву возможностей. Узлы такого дерева можно рассматривать как совокупности конфигураций, а каждый пото-

мок узла и представляет некоторое подмножество конфигураций. Наконец, каждый лист представляет отдельную конфигурацию или решение соответствующей задачи; каждую такую конфигурацию можно оценить и попытаться выяснить, не является ли она наилучшей среди уже найденных решений.

Если просмотр организован достаточно рационально, каждый из потомков некоторого узла будет представлять намного меньше конфигураций, чем соответствующий узел, поэтому, чтобы достичь листьев, не придется забираться слишком глубоко. Чтобы такая организация поиска не показалась читателю чересчур запутанной, рассмотрим конкретный пример.

Пример 10.8. Вспомните задачу коммивояжера, которую мы рассмотрели в одном из предыдущих разделов. В связи с этой задачей мы описали так называемый "жадный алгоритм", позволяющий найти хороший, хотя и необязательно оптимальный, маршрут. Посмотрим, как можно было бы найти оптимальный маршрут путем систематического просмотра всех маршрутов. Это можно было бы сделать, рассмотрев все перестановки узлов и определив маршрут, который проходит через узлы в соответствующей последовательности, учитывая наилучший из найденных до сих пор маршрутов. Время, которое потребуется на реализацию такого подхода на графе с n узлами, равняется $O(n!)$, поскольку необходимо рассмотреть $(n - 1)!$ различных перестановок,[1] а оценка каждой перестановки занимает время $O(n)$.

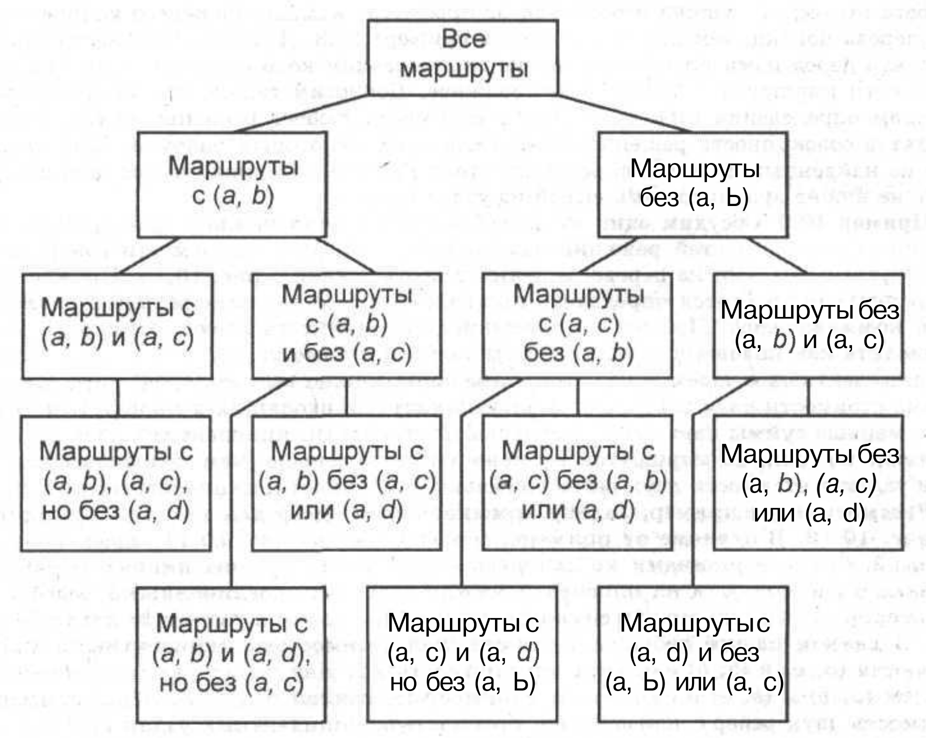


Рис. 10.15. Начало дерева решений для задачи коммивояжера

Мы рассмотрим несколько иной подход, который в наихудшем случае оказывается ничуть не лучше описанного выше, однако в среднем — в сочетании с *методом ветвей и границ,* который мы далее кратко рассмотрим, — позволяет получить ответ

[1] Обратите внимание: нам не нужно рассматривать все n! перестановок, поскольку исходный пункт маршрута не имеет значения. Следовательно, мы можем рассматривать только те перестановки, которые начинаются с 1.

намного быстрее, чем метод "перебора всех перестановок". Построение дерева начинается с корня, который представляет все маршруты. Маршруты соответствуют уже упоминавшимся выше "конфигурациям". Каждый узел имеет двух сыновей, и маршруты, которые представляет узел, делятся этими сыновьями на две группы: которые содержат определенное ребро и у которых такого ребра нет. Например, на рис. 10.15 показаны фрагменты дерева для задачи коммивояжера из рис. 10.11.

Будем проходить ребра дерева решений из рис. 10.15 в лексикографическом порядке: (a, *Ь), (a, c), ... , (a, f), (Ь, c), ... , (b, f), (c, d)* и т.д. Можно, разумеется, выбрать любой другой порядок. Учтите, что не у каждого узла в этом дереве есть два сына. Некоторых сыновей можно удалить, поскольку выбранные ребра не образуют часть маршрута. Таким образом, не существует узлов для маршрутов, содержащих ребра (a, *Ь),* (a, c) и (a, *d),* так как вершина *a* имела бы степень 3 и полученный результат не был бы маршрутом. Точно так же, спускаясь вниз по дереву, можно удалить некоторые узлы, поскольку какой-то город будет иметь степень меньше 2. Например, нет ни одного узла для маршрутов без (a, 6), (a, c), *(a, d)* или (a, *e).* П

Ограничения эвристических алгоритмов

Воспользовавшись идеями, подобными тем, на которых основывается метод альфа-бета отсечения, можно избавиться от просмотра намного большего количества узлов дерева поиска, чем предполагается в примере 10.8. Для определенности представим, что перед нами поставлена задача минимизации некоторой функции, например стоимости маршрута в задаче коммивояжера. Допустим также, что мы располагаем методом определения нижней границы стоимости любого решения из тех, которые входят в совокупность решений, представленных некоторым узлом га. Если наилучшее из найденных до сих пор решений стоит меньше, чем нижняя граница для узла л, то не нужно анализировать любой из узлов ниже п.

Пример 10.9. Обсудим один из способов определения нижней границы для определенных совокупностей решений задачи коммивояжера, причем эти совокупности представлены узлами на дереве решений в соответствии с рис. 10.15. Прежде всего, допустим, что требуется определить нижнюю границу для всех решений данной задачи коммивояжера. Примем во внимание, что стоимость любого маршрута можно вычислить как половину суммы по всем узлам га стоимости пар ребер этого маршрута, инцидентных с узлом га. Из этого замечания можно вывести следующее правило. Сумма стоимости двух ребер, инцидентных узлу га и входящих в маршрут, не может быть меньше суммы двух ребер наименьшей стоимости, инцидентных узлу *n.* Таким образом, ни один из маршрутов не может стоить меньше, чем половина суммы по всем узлам *n* стоимости двух ребер наименьшей стоимости, инцидентных узлу га.

Рассмотрим, например, задачу коммивояжера, граф для которой представлен на рис. 10.16. В отличие от примера, показанного на рис. 10.11, здесь мера расстояний между вершинами не **является** евклидовой, т.е. она никоим образом не связана с расстоянием на плоскости между городами, соединенными ребром. Такой мерой стоимости может служить, например, время в пути или плата за проезд. В данном случае ребрами с наименьшей стоимостью, инцидентными узлу a, являются (a, *d)* и (a, *Ь)* с суммарной стоимостью 5. Для узла ***b*** такими ребрами являются (a, *Ъ)* и ***(b, e)*** с суммарной стоимостью, равной 6. Аналогично, суммарная стоимость двух ребер с наименьшей стоимостью, инцидентных узлам *c, d и e,* равняется соответственно 8, 7 и 9. Нижняя граница стоимости маршрута составит, таким образом, **(5+6+8+7+9)/2=17.5.**

Допустим теперь, что нужно получить нижнюю границу стоимости для некоторого подмножества маршрутов, определяемого некоторым узлом на дереве поиска. Если это дерево поиска выглядит так, как в примере 10.8, то каждый узел представляет маршруты, определяемые совокупностью ребер, которые должны входить в этот маршрут, и совокупностью ребер, которые могут не входить в этот маршрут. Эти *ограничения* сказываются на том, какие два ребра с наименьшей стоимостью мы выбираем в каждом узле. Очевидно, что ребро, на которое накладывается ограничение,

обусловливающее наличие этого ребра в каком-либо маршруте, необходимо включить между двумя выбранными ребрами независимо от того, имеют ли они наименьшую стоимость или стоимость, ближайшую к **наименьшей.**[1] Аналогично, ребро, на которое накладывается ограничение, исключающее его из маршрута, выбрать нельзя, даже если у него наименьшая стоимость.

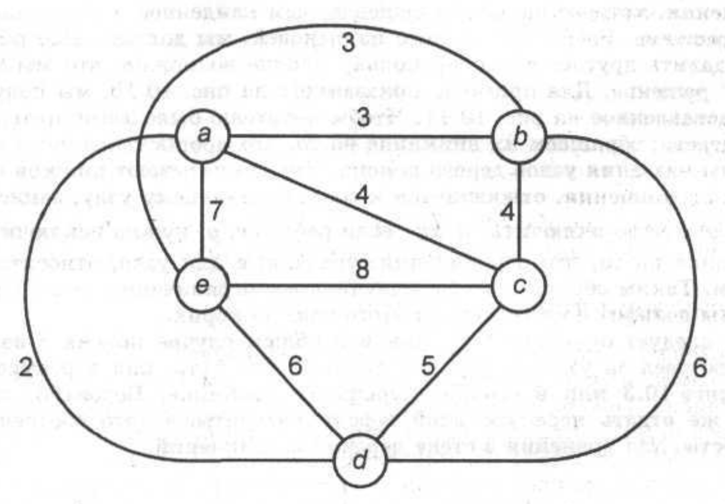


Рис. 10.16. Граф для задачи коммивояжера

Таким образом, если ограничения обязывают включить в маршрут ребро *(а, е)* и исключить ребро *(Ь, с),* то двумя ребрами для узла *а* будут (а, *а)* и (а, *е),* общая стоимость которых равна 9. Для узла *Ь* мы выбираем ребра (а, *b*) и (6, *е),* общая стоимость которых, как и раньше, равняется 6. Для узла *с* мы не можем выбрать ребро (*b, с)* и поэтому выбираем ребра (а, с) и (с, *d),* общая стоимость которых равна 9. Для узла *d* мы, как и раньше, выбираем ребра (а, *d)* и (с, *d),* тогда как для узла *е* мы должны выбрать ребро (а, *е)* и еще выбираем ребро *(Ь, е).* Таким образом, нижняя граница при ограничениях равняется **(9+6+9+7+10)/2** = 20.5. □

Построим дерево поиска вдоль **путей,** предложенных в примере 10.8. Как и в примере, будем рассматривать ребра в лексикографическом порядке. Каждый раз, когда есть *разветвление* для двух сыновей какого-либо узла, мы пытаемся **принять** дополнительные решения относительно того, какие ребра необходимо включить или исключить из маршрутов, представленных соответствующими узлами. Ниже перечислены правила, которые будем использовать для принятия этих решений.

1. Если исключение ребра *(х, у)* приводит к тому, что у вершины *х* или *у* нет других инцидентных ребер на данном маршруте, тогда ребро *(х, у)* необходимо включить.
2. Если включение ребра *(х, у)* приводит к тому, что у вершины *х* или *у* будет более двух ребер на данном маршруте или образуется цикл (не совпадающий с маршрутом) с ранее включенными ребрами, тогда ребро *(х, у)* необходимо исключить.

При разветвлении вычисляем нижние границы для обоих сыновей. Если нижняя граница для какого-нибудь из сыновей оказывается не ниже, чем у найденного на данный момент маршрута с наименьшей стоимостью, мы можем отсечь этого сына и не рассматривать его потомков. Интересно отметить, что бывают ситуации, когда нижняя граница для узла *n* оказывается ниже, чем наилучшее из найденных на

[1] Правила построения дерева поиска предусматривают исключение любой совокупности ограничений, которая не позволяет получить какой-либо маршрут (например, по той причине, что этому маршруту должны принадлежать три ребра, инцидентных одному узлу).

данный момент решений; тем не менее, обоих сыновей узла *n* можно исключить, поскольку их нижние границы оказываются выше стоимости наилучшего из найденных на данный момент решений.

Если ни одного из сыновей удалить невозможно, мы применяем эвристический прием, рассматривая сначала сына с меньшей нижней границей, в надежде быстрее достичь решения, которое окажется дешевле, чем найденное к настоящему времени наилучшее решение. Рассмотрев одного из сыновей, мы должны еще раз проверить, нельзя ли удалить другого сына, поскольку вполне возможно, что мы нашли новое "наилучшее" решение. Для примера, показанного на рис. 10.16, мы получаем дерево поиска, представленное на рис. 10.17. Чтобы читателю было легче интерпретировать узлы этого дерева, обращаем их внимание на то, что прописными буквами на рисунке обозначены названия узлов дерева поиска. Числа указывают нижние границы; мы перечисляем ограничения, относящиеся к соответствующему узлу, записывая *xy*, если ребро *(x, y)* нужно включить, и $\overline{xy}$, если ребро *(x, y)* нужно исключить. Обратите также внимание на то, что ограничения, указанные для узла, относятся и ко всем его потомкам. Таким образом, чтобы получить все ограничения, относящиеся к данному узлу, мы должны пройти путь от этого узла до корня.

Наконец, следует отметить, что, как и в общем случае поиска с возвратом, мы строим дерево узел за узлом, сохраняя только один путь, как в рекурсивном алгоритме листинга 10.3 или в его нерекурсивном двойнике. Вероятно, предпочтение следует все же отдать нерекурсивной версии алгоритма — это обеспечит дополнительное удобство для хранения в стеке перечня ограничений.

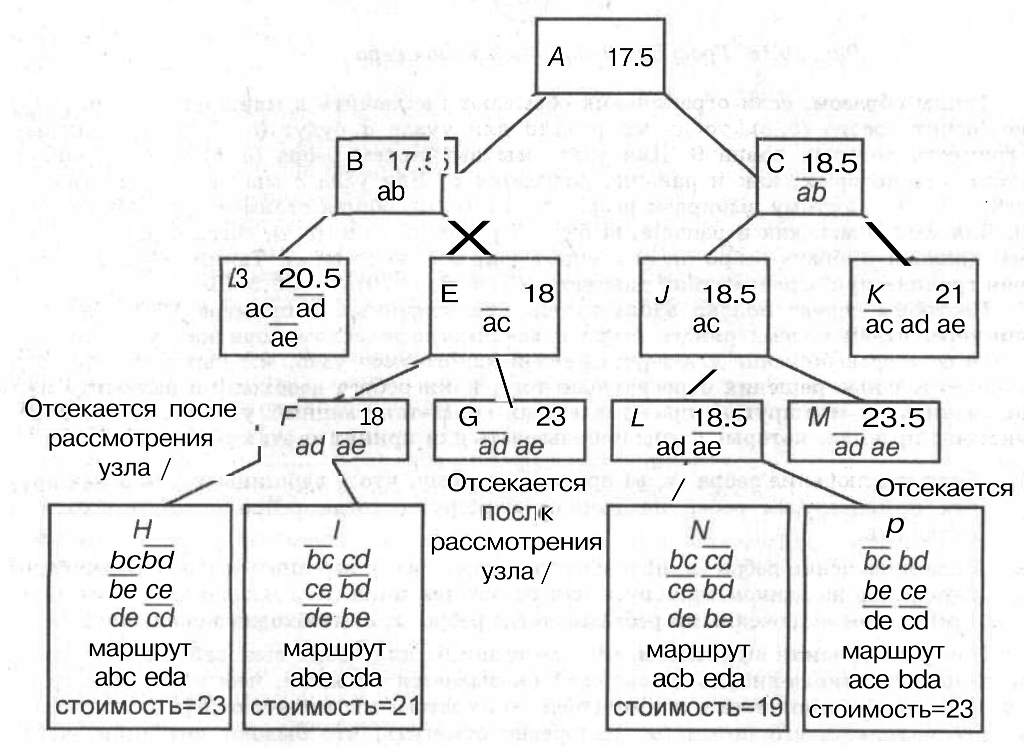


Рис. 10.17. Дерево поиска для задачи коммивояжера

Пример 10.10. На рис. 10.17 показано дерево поиска для задачи коммивояжера, соответствующей графу из рис. 10.16. Покажем, как строится это дерево. Принимаем за исходную точку корень *A*. Первым ребром в лексикографическом порядке бу-

дет ребро (а, *b*), поэтому мы рассматриваем двух его сыновей *B* и *C*, соответствующих ограничениям ab и $\overline{ab}$ соответственно. Пока еще у нас нет наилучшего на данный момент решения, поэтому придется рассматривать как узел *B*, так и узел *C*.[1] Включение ребра (а, *Ъ)* в маршрут не повышает его нижнюю границу, но исключение этого ребра повышает ее до 18,5, поскольку два самых дешевых "законных" ребра для узлов а и *Ь* теперь стоят соответственно 6 и 7 (в сравнении с 5 и 6 при отсутствии ограничений). В соответствии с нашей эвристикой мы рассмотрим сначала потомков узла *B*.

Следующим ребром в лексикографическом порядке будет ребро (а, с). Таким образом, мы переходим к узлам *D* и *E*, соответствующим маршрутам, где ребро (а, *с)* соответственно включается или исключается. Применительно к узлу *D* можно сделать вывод, что ни ребро (а, d), ни ребро (а, *е)* не могут входить в маршрут (в противном случае узел а имел бы слишком много инцидентных ребер). В соответствии с нашим эвристическим подходом мы рассматриваем сначала узел *E*, а затем *D* и переходим к ребру (о, *d)*. Нижние границы для узлов *F* и *G* равняются соответственно 18 и 23. Для каждого из этих узлов нам известны три ребра из ребер, инцидентных *а*, поэтому мы можем сделать определенные выводы относительно оставшегося ребра *(а, е)*.

Рассмотрим сначала сыновей узла *F*. Первым оставшимся ребром в лексикографическом порядке будет ребро *(Ь, с)*. Если мы включим в маршрут это ребро, то не сможем включить ребро *(b, d)* или *(Ь, е)*, так как уже включили ребро (а, *b*). Поскольку мы уже исключили ребра (а, *е)* и *(Ь, е)*, у нас должны быть ребра *(с, е)* и (d, *е)*. Ребро *(с, d)* не может входить в маршрут, иначе у вершин *с* и *d* три инцидентных ребра входили бы в маршрут. Остается один маршрут (а, *Ь, с, е, d, а)*, стоимость которого равна 23. Аналогично можно доказать, что узел *I* с исключенным ребром *(Ь, с)* представляет лишь маршрут *(а, Ь, е, с, d, а)*, стоимость которого равняется 21. На данный момент это маршрут с наименьшей стоимостью.

Теперь возвратимся в узел *E* и рассмотрим его второго сына, узел *G*. Но нижняя граница G равняется 23, что превосходит наилучшую на данный момент стоимость — 21. Следовательно, мы удаляем узел *G*. Теперь возвращаемся в узел *B* и исследуем его второго сына, узел *D*. Нижняя граница для *D* равняется 20.5, но, так как стоимости являются целочисленными значениями, нам известно, что ни у одного из маршрутов, включающих *D*, стоимость не может быть меньше 21. Поскольку у нас уже имеется столь дешевый маршрут, нам не придется рассматривать потомков *D*, следовательно, отсекаем узел *D*. Теперь возвращаемся в узел *A* и рассматриваем его второго сына, узел *C*.

На уровне узла С мы рассмотрели только одно ребро (а, *b*). Узлы *J* и *K* являются сыновьями узла С. Узел *J* соответствует тем маршрутам, которые содержат ребро (а, *с)*, но не содержат ребро (а, *Ь)*, его нижняя граница равняется 18,5. Узел *K* соответствует тем маршрутам, которые не содержат ни ребро (а, с), ни ребро (а, *Ь)*, отсюда следует вывод, что эти маршруты содержат ребра (а, d) и (а, *е)*. Нижняя граница для узла *K* равна 21, поэтому можем отсечь узел *K*, поскольку уже известен маршрут с такой стоимостью.

Далее рассматриваем сыновей узла *J*, которыми являются узлы *L* и *M*; мы отсекаем узел *M*, поскольку его нижняя граница превосходит стоимость наилучшего из найденных на данный момент маршрутов. Сыновьями узла *L* являются узлы *N* и *P*, соответствующие маршрутам, которые содержат ребро *(b, с)* и исключают ребро *(Ъ, с)*. Учитывая степень вершин *b* и с и помня о том, что отброшенные ребра не могут образовывать цикл из менее чем всех пяти вершин, можем сделать вывод, что узлы *N и P* (каждый из них) представляют отдельные маршруты. Один из них (а, *с, b, е, d, а)* имеет наименьшую среди всех маршрутов стоимость — 19. Таким образом, мы проверили все дерево и частично сократили его. D

---

[1] Мы могли бы взять за исходную точку какое-либо решение, найденное эвристическим способом (например, с помощью "жадного" алгоритма), хотя для нашего примера это не имеет значения. Стоимость "жадного" решения для графа из рис. 10.16 равняется 21.

10.5. Алгоритмы локального поиска

Описанная ниже стратегия нередко приводит к оптимальному решению задачи.

1. Начните с произвольного решения.
2. Для улучшения текущего решения примените к нему какое-либо преобразование из некоторой заданной совокупности преобразований. Это улучшенное решение становится новым "текущим" решением.
3. Повторяйте указанную процедуру до тех пор, пока ни одно из преобразований в заданной их совокупности не позволит улучшить текущее решение.

Результирующее решение может, хотя и необязательно, оказаться оптимальным. В принципе, если "заданная совокупность преобразований" включает все преобразования, которые берут в качестве исходного одно решение и заменяют его каким-либо другим, процесс "улучшений" не закончится до тех пор, пока мы не получим оптимальное решение. Но в таком случае время выполнения пункта (2) окажется таким же, как и время, требующееся для анализа всех решений, поэтому описываемый подход в целом окажется достаточно бессмысленным.

Этот метод имеет смысл лишь в том случае, когда мы можем ограничить нашу совокупность преобразований небольшим ее подмножеством, что дает возможность выполнить все преобразования за относительно короткое время: если "размер" задачи равняется га, то мы можем допустить $O(n^2)$ или $O(n^3)$ преобразований. Если совокупность преобразований невелика, естественно рассматривать решения, которые можно преобразовывать одно в другое за один шаг, как "близкие". Такие преобразования называются "локальными", а соответствующий метод называется *локальным поиском.*

Пример 10.11. Одной из задач, которую можно решить именно методом локального поиска, является задача нахождения минимального остовного дерева. Локальными преобразованиями являются такие преобразования, в ходе которых мы берем то или иное ребро, не относящееся к текущему остовному дереву, добавляем его в это дерево (в результате мы должны получить цикл), а затем убираем из этого цикла в точности одно ребро (предположительно, ребро с наивысшей стоимостью), чтобы образовать новое дерево.

Рассмотрим, например, граф на рис. 10.16. Мы можем начать с дерева, показанного на рис. 10.18,а. Одно из преобразований, которые можно было бы выполнить, заключается в добавлении ребра *(d, e)* и устранении из образовавшегося цикла *(e, a, c, d, e)* какого-либо другого ребра. Если мы удалим ребро (a, *e),* то уменьшим стоимость дерева с 20 до 19. Это преобразование можно выполнить, получив в результате дерево (рис. 10.18,б), к которому мы опять попытаемся применить улучшающее преобразование. Одно из таких преобразований сводится к вставке ребра (a, *d)* и удалению из образовавшегося цикла ребра *(c, d).* Результат этого преобразования показан на рис. 10.18,в. Затем можно вставить ребро (a, *b)* и убрать ребро *(b, c),* как показано на рис. 10.18,г, а потом — вставить ребро (b, e) вместо ребра *(d, e).* Результирующее дерево (рис. 10.18,д) является минимальным. Мы можем убедиться в том, что каждое ребро, не входящее в состав этого дерева, имеет наивысшую стоимость среди всех ребер в цикле, который они с этим ребром составляют. Таким образом, к дереву на рис. 10.18,гпреобразования уже неприменимы. D

Время, которое занимает выполнение алгоритма в примере 10.11 на графе из *n* узлов и *e* ребер, зависит от количества требующихся улучшений решения. Одна лишь проверка того факта, что преобразования уже неприменимы, может занять $O(ne)$ времени, поскольку для этого необходимо перебрать *e* ребер, а каждое из них может образовать цикл длиной примерно *n.* Таким образом, этот алгоритм несколько хуже, чем алгоритмы Прима и Крускала, однако он может служить примером получения оптимального решения на основе локального поиска.

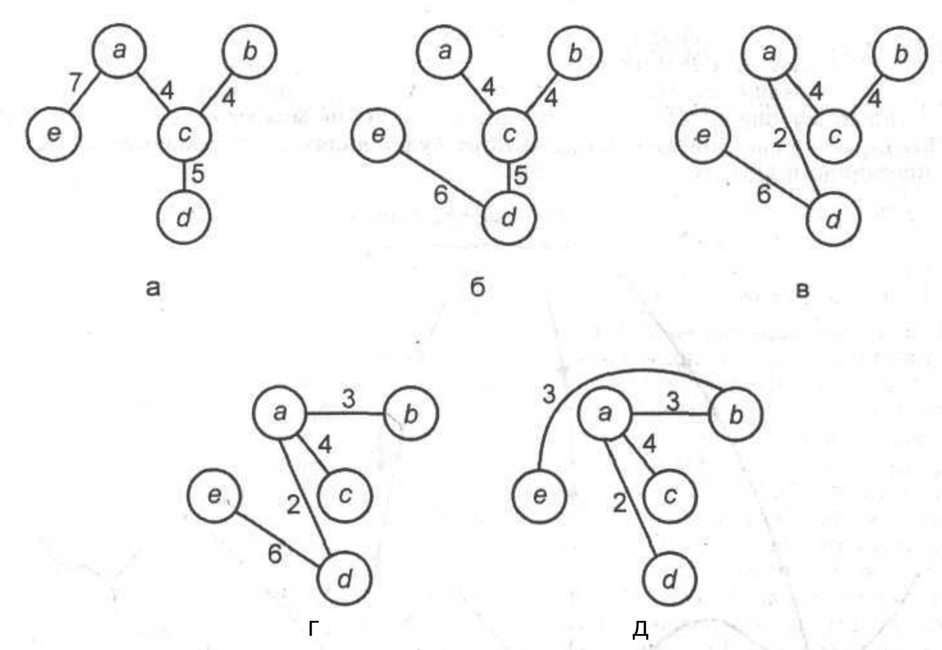


*Рис. 10.18. Локальный поиск минимального **остовного** дерева*

Локальные и глобальные оптимальные решения

Алгоритмы локального поиска проявляют себя с наилучшей стороны как эвристические алгоритмы для решения задач, точные решения которых требуют экспоненциальных затрат времени. Общепринятый метод поиска состоит в следующем. Начать следует с ряда произвольных решений, применяя к каждому из них локальные преобразования до тех пор, пока не будет получено ***локально-оптимальное*** решение, т.е. такое, которое не сможет улучшить ни одно преобразование. Как показано на рис. 10.19, на основе большинства (или даже всех) произвольных начальных решений мы нередко будем получать разные локально-оптимальные решения. Если нам повезет, одно из них окажется ***глобально-оптимальным***, т.е. лучше любого другого решения.

На практике мы можем и не найти глобально-оптимального решения, показанного на рис. 10.19, поскольку количество локально-оптимальных решений может оказаться колоссальным. Однако мы можем по крайней мере выбрать локально-оптимальное решение, имеющее минимальную стоимость среди всех найденных нами решений. Поскольку количество видов локальных преобразований, использующихся для решения различных задач, очень велико, мы завершим этот раздел описанием двух примеров: задачи коммивояжера и простой задачи размещения (коммутации) блоков.

Задача коммивояжера

Методы локального поиска особенно хорошо подходят для решения задачи коммивояжера. Простейшим преобразованием, которым можно в этом случае воспользоваться, является так называемый "двойной выбор". Он заключается в том, что мы выбираем любые два ребра, например ребра (A, B) и (C, D), показанные на рис. 10.20, удаляем их и "перекоммутируем" соединявшиеся ими точки так, чтобы образовался новый маршрут. На рис. 10.20 этот новый маршрут начинается в точке

B, продолжается по часовой стрелке до C, проходит по ребру (C, *A),* затем — против часовой стрелки от *A* к *D* и наконец по ребру *(D, B).* Если сумма длин (A, C) и *(B, D)* оказывается меньше суммы длин *(A, B)* и (C, *D),* значит, нам удалось получить улучшенный **маршрут.**[1] Обратите внимание, что мы не можем соединить точки *A* и *D, B* и C, поскольку полученный результат будет являться не маршрутом, а двумя изолированными друг от друга циклами.

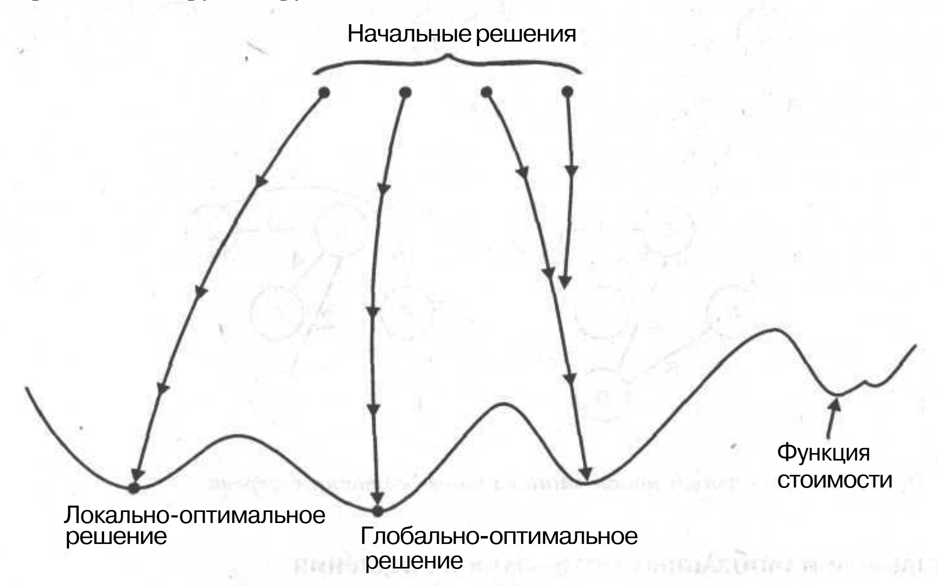


Рис. 10.19. Локальный поиск в пространстве решений

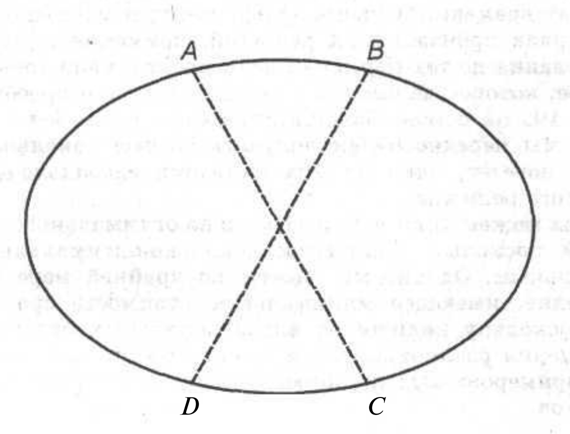


Рис. 10.20. *Двойной выбор*

[1] Вас не должен вводить в заблуждение рис. 10.20. Действительно, если длины ребер являются расстояниями на плоскости, тогда ребра, показанные пунктирными линиями на этом рисунке, должны быть длиннее ребер, которые мы удалили. **Однако,** вообще говоря, нет никаких оснований предполагать, что расстояния, **показанные** на рис. 10.20, обязательно должны быть евклидовыми расстояниями, но если они и являются таковыми, то пересекающимися могли бы быть ребра *(A, B)* и (C, *D),* а не *(A, C)* и (B, *D).*

Чтобы найти локально-оптимальный маршрут, мы начинаем с какого-либо произвольного маршрута и рассматриваем все пары несмежных ребер, такие как *(A, B)* и (C, *D)* на рис. 10.20. Если данный маршрут можно улучшить путем замены этих ребер на *(A, C)* и *(B, D),* это нужно сделать, а затем продолжить рассмотрение оставшихся пар ребер. Обратите внимание, что каждое из новых ребер (A, C) и *(B, D)* должно образовать пару со всеми другими ребрами данного маршрута, результатом чего могут явиться дополнительные улучшения.

Пример 10.12. Вернемся к рис. 10.16 и допустим, что в качестве исходного выбран маршрут, показанный на рис. **10.21,а**. Ребра (a, *e)* и (c, *d*) общей стоимостью 12 можно заменить ребрами (a, *d)* и (e, *e)* общей стоимостью 10, как показано на рис. **10.21,б**. Затем ребра (a, *Ь)* и (c, *e)* можно заменить на ребра (a, c) и *(b, e),* что обеспечило бы нам оптимальный маршрут, показанный на рис. **10.21,в**. Легко убедиться, что на этом рисунке нельзя удалить ни одну пару ребер, выгодно заменив ее пересекающимися ребрами с теми же конечными точками. Возьмем хотя бы один пример: ребра (*b, c)* и *(d, e)* вместе имеют относительно высокую стоимость — 10. Но (*c, e)* и *(b, d)* еще хуже, поскольку их совместная стоимость равна 14. □

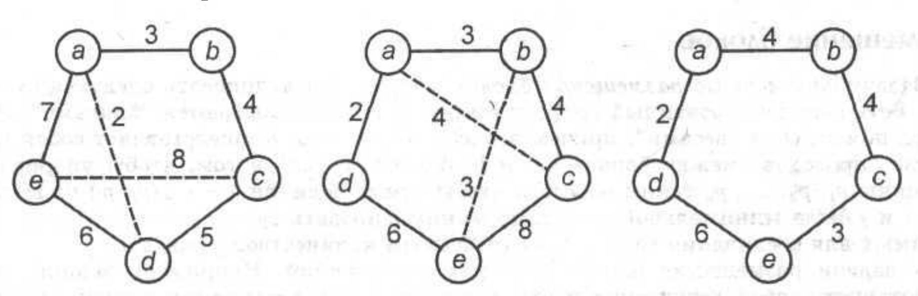


Рис. 10.21. Оптимизация решения задачи коммивояжера посредством двойного выбора

Двойной выбор можно обобщить как выбор *k* элементов для любого постоянного ft. В этом случае мы удаляем до *k* ребер и "перекоммутируем" оставшиеся элементы в любом порядке, пытаясь получить какой-либо маршрут. Обратите внимание: мы не требуем, чтобы удаленные ребра вообще были несмежными, хотя в случае двойного выбора не было никакого смысла рассматривать удаление двух смежных ребер. Обратите также внимание, что при $k > 2$ существует несколько способов соединения частей графа. На рис. 10.22 представлен процесс тройного выбора с помощью любой из перечисленных ниже восьми совокупностей ребер.

1. *(A. F), (D, E), (B, C)* (исходный маршрут)
2. *(A, F), (C, E), (D, B)* (двойной выбор)
3. *(A, E), (F, D), (B, C)* (еще один двойной выбор)
4. (A, *E), (F,* C), *(B, D)* (тройной выбор)
5. *(A, D), (C, E), (B, F)* (еще один тройной выбор)
6. *(A, D), (C, F), (B, E)* (еще один тройной выбор)
7. *(A, C), (D, E), (B, F)* (двойной выбор)
8. *(A, C), (D, F), (B, E)* (тройной выбор)

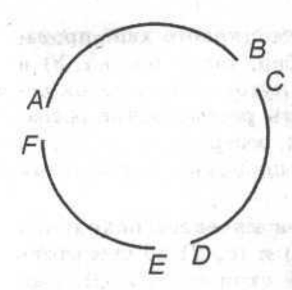


Рис. 10.22. Выбор элементов маршрута после удаления трех ребер '

Легко убедиться в том, что для фиксированного k количество различных преобразований при fe-выборе, которые необходимо рассмотреть при наличии n вершин, равно $O(n^k)$. Например, при $k = 2$ точное количество таких преобразований равно $n(n - 3)/2$. Однако время, которое требуется для получения какого-либо из локально-оптимальных маршрутов, может оказаться значительно больше этой величины, поскольку для получения этого локально-оптимального маршрута надо выполнить довольно много локальных преобразований, а каждое улучшающее преобразование связано с введением новых ребер, которые могут участвовать в последующих преобразованиях и еще больше улучшают данный маршрут. В работе [68] показано, что с практической точки зрения "выбор с переменной глубиной" (т.е. выбор с разными значениями k на разных этапах) является чрезвычайно эффективным методом и с большой вероятностью обеспечивает получение оптимального маршрута для задач коммивояжера с 40-100 городами.

Размещение блоков

Задачу *одномерного размещения блоков* можно сформулировать следующим образом. Есть неориентированный граф, вершины которого называются "блоками". Ребра графа помечаются "весами", причем вес $w(a, b)$ ребра (а, *Ь)* представляет собой количество "проводов" между блоками а и Ъ. Задача состоит в том, чтобы упорядочить вершины $p_1, p_2 \ldots p_n$ таким образом, чтобы сумма величин $|i - j|\ w(p_i, p_j)$ по всем парам i и j была минимальной, т.е. надо минимизировать сумму длин проводов, необходимых для соединения всех блоков требуемым количеством проводов.

У задачи размещения блоков есть ряд приложений. Например, "блоки" могут представлять собой логические платы в стойке; весом соединения между платами в этом случае является количество соединяющих их проводников. Аналогичная задача возникает и в случае проектирования интегральных схем на основе набора стандартных модулей и взаимосвязей между ними. Более общий случай задачи одномерного размещения блоков допускает размещение "блоков", имеющих определенную высоту и ширину, в двумерной области с минимизацией суммы длин проводов, соединяющих эти "блоки". Эта задача, имеющая множество различных приложений, применяется и в проектировании интегральных схем.

Для нахождения локальных оптимальных решений задач одномерного размещения блоков можно применять ряд локальных преобразований. Вот некоторые из них.

1. Произвести взаимную перестановку смежных блоков p_i и p_{i+1}, если результирующий порядок имеет меньшую стоимость. Пусть $L(j)$ — сумма весов ребер, расположенных слева от блока p_j, т.е. $\sum_{k=1}^{j-1} w(p_k, p_j)$. Аналогично обозначим через $R(j)$ сумму $\sum_{k=j+1}^{n} w(p_k, p_j)$ весов ребер, расположенных справа от блока p_j. Улучшение можно выполнить, если $L(i) - R(i) + R(i + 1) - L(i + 1) + 2w(p_i, p_{i+1})$ является отрицательным числом. Читатель может убедиться в справедливости этой формулы, вычислив соответствующие стоимости до и после взаимной перестановки и оценив разность между этими стоимостями.
2. Взять пакет p_i и вставить его между p_j и p_{j+1} при некоторых значениях i и j.
3. Выполнить взаимную перестановку двух блоков p_i и p_j.

Пример 10.13. Возьмем в качестве примера размещения блоков граф, показанный на рис. 10.16. Ограничимся простой совокупностью преобразований (1). Начальное размещение блоков *a. Ъ, c, d, e* показано на рис. **10.23,a**, его стоимость равна 97. Обратите внимание, что функция стоимости определяет вес ребра в со-

ответствии с количеством промежуточных ребер между блоками в выбранном размещении блоков, соединенных этим **ребром,** поэтому "вклад" ребра *(a, e)* в общую стоимость составляет $4 \times 7 = 28$. Рассмотрим взаимную перестановку блоков *d* и *e*. Сейчас $L(d) = 13$, $R(d) = 6$, $L(e) = 24$ и $R(e) = 0$. Таким образом, $L(d) - R(d) + R(e) - L(e) + 2w(d, e) = -5$, поэтому следует произвести перестановку блоков *d и e,* получив улучшенное размещение *a, Ь, c, e, d,* стоимость которого, как следует из **рис. 10.23,б,** равна 92.

На рис. **10.23,б** можно произвести выгодную взаимную перестановку блоков ***c* и *e*,** получив таким образом размещение блоков, показанное на рис. **10.23,в;** стоимость этого размещения равна 91. Размещение на рис. **10.23,в** является локально-оптимальным для совокупности преобразований (1). Однако оно не является глобально-оптимальным, поскольку для размещения *a, c, e, d, b* стоимость равна 84. □

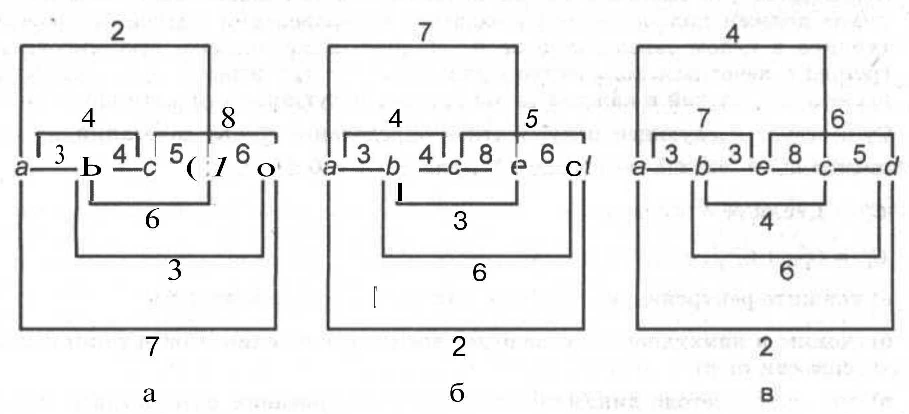


Рис. *10.23. Локальные оптимизации*

Как и в задаче коммивояжера, мы не в состоянии точно оценить время, необходимое для достижения локального оптимума. Можно лишь констатировать факт, что для совокупности преобразований (1) надо рассмотреть только п - 1 преобразований. Далее, после того как мы однажды вычислили *L(i)* и *R(i),* нам придется лишь заменять их при выполнении взаимной перестановки p_i с p_{i-1} или p_{i+1}. Более того, подобный пересчет не представляет особого труда. Если, например, выполняется взаимная перестановка блоков p_i и p_{i+1}, тогда новыми значениями ДО и *R(i)* будут соответственно величины $L(i+1) - w(p_i, p_{i+1})$ и $R(i+1) + w(p_i, p_{i+1})$. Таким образом, времени порядка $O(n)$ будет достаточно, чтобы проверить, является ли выполняемое преобразование улучшающим, и повторно вычислить все *Щ)* и $R(i)$. Кроме того, на начальное вычисление всех значений *Щ)* и *R(i)* также понадобится лишь $O(n)$ времени, если мы воспользуемся рекуррентными соотношениями

$$L(1) = 0\,,$$
$$L(i) = L(i-1) + w(p_{i-1}, p_i)$$

и аналогичными рекуррентными соотношениями для *R*.

Для сравнения: каждая из совокупностей преобразований (2) и (3) имеет $O(n^2)$ членов. Таким образом, потребуется $O(n^2)$ времени лишь для того, чтобы убедиться в том, что мы вышли на одно из локально-оптимальных решений. Однако, как и в случае совокупности преобразований (1), мы не в состоянии точно оценить общее время, которое может потребоваться для выполнения ряда последовательных улучшений, поскольку каждое такое улучшение может создавать дополнительные возможности для дальнейших улучшений.

Упражнения

10.1. Сколько перемещений необходимо выполнить для перестановки п дисков в задаче о "Ханойских башнях"?

***10.2.** Докажите, что рекурсивный алгоритм декомпозиции, используемый для решения задачи о "Ханойских башнях", и нерекурсивный алгоритм, описанный в начале раздела **10.1,** выполняют одни и те же действия.

10.3. Используя алгоритм декомпозиции (листинг 10.1), выполните умножение чисел 1011 и 1101.

***10.4.** Обобщите составление расписания теннисного турнира (см. раздел 10.1), если количество его участников не выражается степенью числа 2. *Совет.* Если *n* (количество участников турнира) нечетное, тогда каждый день один из участников должен полностью освобождаться от проведения матчей, а проведение турнира в целом займет *n,* а не *n* — 1 дней. Если, однако, сформировать две группы с нечетным количеством **участников,** тогда игроки, освобожденные от проведения матчей в каждой такой **группе,** могут провести матч друг с другом.

10.5. Существует следующее рекуррентное определение числа сочетаний из *n* элементов по *m,* обозначается как С™ , для л > 1 и 0 < *m* < га:

С™ = **1,**если $m = 0$ или $m = n$,

$C_n^m = C_{n-1}^{n}1 + {}_nC_{n-1}^{m-1}$, если $0 < m < n$.

а) укажите рекурсивную функцию для вычисления чисел С™ ;

б) каким в наихудшем случае будет время вычисления этой функции в зави**симости** от га?

в) на основе метода динамического программирования разработайте алгоритм для вычисления чисел С™ . *Совет.* Этот алгоритм должен составлять таблицу, известную как треугольник Паскаля;

г) каким будет время выполнения этого алгоритма в зависимости от га?

10.6. Один из способов вычисления количества сочетаний из *n* элементов по *m* заключается в вычислении выражения $(n)(n - 1)(n - 2)...(n - m + 1)/(1)(2)...(m)$.

а) каким в наихудшем случае будет время вычисления этого алгоритма в зависимости от п?

*б) возможно ли примерно таким же способом вычислить вероятности *P(i, j)* победы в турнирах (см. раздел 10.2)? Как быстро можно выполнить это вычисление?

10.7. а) перепишите программу вычисления вероятностей, показанную в листинге 10.2, если вероятность выигрыша первой командой любой игры равняется p;

б) если первая команда выиграла один матч, а вторая — два, и вероятность выигрыша 1-й командой любой игры равна 0,6, то у какой из команд больше шансов выиграть весь турнир?

10.8. Программе вычисления вероятностей (листинг 10.2) требуется объем памяти порядка $O(n^2)$. Перепишите эту программу так, чтобы ей требовалось только $O(n)$ объема памяти.

***10.9.** Докажите, что вычисление решения уравнения (10.4) требует ровно $2C^i_{i+j} - 1$ вычислений различных значений *P.*

10.10. Найдите минимальную триангуляцию для правильного восьмиугольника в предположении, что расстояния вычисляются как **евклидовые** расстояния.

10.11. *Задачу разбиения на абзацы* в простейшей форме можно сформулировать следующим образом. Дана последовательность слов $w_1, w_2, \ldots, w_k$ длиной $l_1, l_2, \ldots, l_k$, которые нужно разбить на строки длиной L. Слова разделяются пробелами, стандартная ширина которых равна b, но пробелы при необходимости могут удлиняться или сжиматься (но без "наползания" слов друг на друга) так, чтобы длина строки $w_i\ w_{i+1} \ldots w_j$ равнялась в точности L. Однако *штрафом* за такое удлинение или сжатие пробелов является общая величина, на которую пробелы удлиняются или сжимаются, т.е. стоимость формирования строки $w_i w_{i+1} \ldots w_j$ при $j > i$ равна $(j - i)\ b' - b|$, где b' — фактическая ширина пробелов, равная $(L - l_i - l_{i+1} - \ldots - l_j)/(j\text{-} i)$. При $j = k$ (речь идет о последней строке) эта стоимость равна нулю, если только b' не окажется меньше b, поскольку последнюю строку растягивать не требуется. На основе метода динамического программирования разработайте алгоритм для разделения с наименьшей стоимостью последовательности слов $w_1, w_2, \ldots, w_k$ на строки длиной L. *Совет.* Для $i = k, k - 1, \ldots, 1$ вычислите наименьшую стоимость формирования строк $w_i, w_{i+1}, \ldots, w_k$.

10.12. Допустим, есть n элементов $x_1, x_2, \ldots, x_n$, связанных линейным порядком $x_1 < x_2 < \ldots < x_n$, которое нужно представить в виде двоичного дерева поиска. Обозначим p_i вероятность запроса на поиск элемента x_i. Тогда для любого заданного двоичного дерева поиска средняя стоимость поиска составит $\sum_{i=1}^{n} p_i(d_i + 1)$, где d_i — глубина узла, содержащего x_i. При условии, что заданы все вероятности p_i, и предполагая, что x_i никогда не изменяются, можно построить двоичное дерево поиска, минимизирующее стоимость поиска. На основе метода динамического программирования разработайте алгоритм, реализующий такое двоичное дерево поиска. Каким будет время выполнения алгоритма? *Совет.* Вычислите для всех *i я j* оптимальную стоимость поиска среди всех деревьев, содержащих только элементы $x_i, x_{i+1}, \ldots, x_{i+j-1}$, т.е. всего j элементов, начиная с x_i.

****10.13.** Для монет какого достоинства "жадный" алгоритм, описанный в разделе 10.3, обеспечивает оптимальное решение выдачи сдачи?

10.14. Составьте рекурсивный алгоритм триангуляции, обсуждавшийся в разделе 10.2. Покажите, что результатом выполнения этого рекурсивного алгоритма будут ровно 3^{s-4} вызовов в случае нетривиальных задач, если начать с задачи размером $s > 4$.

10.15. Опишите "жадный" алгоритм для

а) задачи одномерного размещения блоков;

б) задачи разбиения на абзацы (упражнение 10.11).

Приведите пример, когда алгоритм не обеспечивает оптимального решения, или покажите, что таких случаев вообще не может быть.

10.16. Постройте нерекурсивную версию алгоритма просмотра дерева, представленного в листинге 10.3.

10.17. Рассмотрим дерево игры, в котором используются шесть шариков и игроки 1 и 2 выбирают по очереди от одного до трех шариков. Игрок, взявший последний шарик, считается проигравшим.

а) составьте полное дерево этой игры;

б) если это дерево игры просматривать с помощью метода альфа-бета отсечений, а узлы, представляющие конфигурации с наименьшим количеством шариков, просматривать первыми, какие узлы будут отсечены?

в) кто выиграет, если оба игрока будут действовать оптимальным образом?

***10.18.** Разработайте алгоритм ветвей и границ для задачи коммивояжера, взяв за основу идею, что маршрут должен начинаться с вершины 1 и ответвляться на каждом уровне, исходя из того, какой узел должен быть в этом маршруте следующим (а не из того, какое конкретное ребро выбрано на рис. 10.17). Что может служить подходящим критерием оценки нижней границы для конфигураций, которые являются списками вершин 1, $v_1, v_2, \ldots$ » определяющих начало маршрутов? Как будет вести себя алгоритм применительно к графу из рис. 10.16, если предположить, что вершина *a* является вершиной 1?

***10.19.** Возможным вариантом алгоритма локального поиска для задачи разбиения на абзацы является применение локальных преобразований, которые перемещают первое слово одной строки на предыдущую строку или последнее слова строки — на последующую. Будет ли в этом случае каждое локально-оптимальное решение являться глобально-оптимальным?

10.20. Если локальные преобразования состоят лишь из двойных выборов, есть ли для графа из рис. 10.16 какие-либо локально-оптимальные маршруты, которые не являются глобально-оптимальными?

Библиографические примечания

Существует множество приложений алгоритмов декомпозиции, включая быстрое преобразование Фурье с временем выполнения $O(n\log n)$, описанное в [21], алгоритм умножения целых чисел с временем выполнения $O(n \log n \log\log n)$, описанный в [96], и алгоритм умножения матриц с временем $O(n^{2,81})$, рассмотренный в [104]. Алгоритм умножения целых чисел с временем выполнения $O(n^{1,59})$ описан в работе [59]. В статье [76] представлено несколько эффективных алгоритмов декомпозиции для модулярной арифметики, а также для полиномиальной интерполяции и оценивания.

С популярным изложением динамического программирования можно ознакомиться в [7]. Применение динамического программирования к триангуляции изложено в [40]. Упражнение 10.11 заимствовано из [66]. В [64] содержится решение задачи построения оптимального дерева двоичного поиска (см. упражнение 10.12).

В [68] описан эффективный эвристический алгоритм для задачи коммивояжера.

Обсуждение NP-полных и других задач, сложных с вычислительной точки зрения, можно найти в монографии [3], а также в работе [41].

ГЛАВА 11

Структуры данных и алгоритмы для внешней памяти

Мы начинаем эту главу с обсуждения различий в характеристиках доступа к устройствам основной (оперативной) и внешней памяти (например, дисководов). Затем представим несколько алгоритмов сортировки файлов данных, хранящихся на устройствах внешней памяти. Завершается глава обсуждением структур данных и алгоритмов, таких как индексированные файлы и B-деревья, которые хорошо подходят для хранения и поиска информации на вторичных устройствах памяти.

11.1. Модель внешних вычислений

В алгоритмах, которые мы обсуждали до сих пор, предполагалось, что объем входных данных позволяет обходиться исключительно основной (оперативной) памятью. Но как быть, если нам нужно, например, отсортировать всех государственных служащих по продолжительности их рабочего стажа или хранить информацию из налоговых деклараций всех граждан страны? Когда возникает необходимость решать подобные задачи, объем обрабатываемых данных намного превышает возможности основной памяти. В большинстве компьютерных систем предусмотрены устройства внешней памяти, такие как жесткие диски, или запоминающие устройства большой емкости, на которых можно хранить огромные объемы данных. Однако характеристики доступа к таким устройствам внешней памяти существенно отличаются от характеристик доступа к основной памяти. Чтобы повысить эффективность использования этих устройств, был разработан ряд структур данных и алгоритмов. В этой главе мы обсудим структуры данных и алгоритмы для сортировки и поиска информации, хранящейся на вторичных устройствах памяти.

В Pascal и некоторых других языках программирования предусмотрен файловый тип данных, предназначенный для представления данных, хранящихся во вторичной памяти. Даже если в языке, которым вы пользуетесь, файловый тип данных не предусмотрен, в операционной системе понятие "внешних" файлов, несомненно, поддерживается. О каких бы файлах мы ни говорили (файлах, предусмотренных в Pascal, или файлах, поддерживаемых непосредственно операционной системой), в любом случае нам придется действовать в рамках ограничений, касающихся способов доступа к файлам. Операционная система делит вторичную память на *блоки* одинакового размера. Размер блока зависит от конкретного типа операционной системы и обычно находится в пределах от 512 до 4096 байт.

Файл можно рассматривать как связанный список блоков, хотя чаще всего операционная система использует древовидную организацию блоков, при которой блоки, составляющие файл, являются листьями дерева, а каждый внутренний узел содержит указатели на множество блоков файла. Если, например, 4 байт достаточно, что-

бы хранить адрес блока, а длина блока составляет 4096 байт, тогда корневой блок может содержать указатели максимум на 1024 блока. Таким образом, файлы, состоящие максимум из 1024 блоков (т.е. примерно четырех миллионов байт), можно представить одним корневым блоком и блоками, содержащими сам файл. Файлы, состоящие из максимум 2^{20} блоков, или 2^{32} байт, можно представить одним корневым блоком, указывающим на 1024 блока промежуточного уровня, каждый из которых указывает на 1024 блока-листа, содержащих определенную часть файла, и т.д.

Базовой операцией, выполняемой по отношению к файлам, является перенос одного блока в *буфер,* находящийся в основной памяти. Буфер представляет собой зарезервированную область в основной памяти, размер которой соответствует размеру блока. Типичная операционная система обеспечивает чтение блоков в том порядке, в каком они появляются в списке блоков, который содержит соответствующий файл, т.е. сначала мы читаем в буфер первый блок файла, затем заменяем его на второй блок, который записывается в тот же буфер, и т.д.

Теперь нетрудно понять концепцию, которая лежит в основе правил чтения файлов в языке Pascal. Каждый файл хранится в виде определенной последовательности блоков; каждый такой блок содержит целое число записей. (Память будет использоваться нерационально, если хранить части одной и той же записи в разных блоках.) Указатель считывания всегда указывает на одну из записей в блоке, который в данный момент находится в буфере. Когда этот указатель должен переместиться на запись, отсутствующую в буфере, настало время прочитать следующий блок файла.

Аналогично, процесс записи файла в языке Pascal можно рассматривать как процесс создания файла в буфере. Когда записи "записываются" в файл, фактически они помещаются в буфер для этого файла — непосредственно вслед за записями, которые уже находятся там. Если очередная запись не помещается в буфер целиком, содержимое буфера копируется в свободный блок вторичной памяти, который присоединяется к концу списка блоков для данного файла. После этого можно считать, что буфер свободен для помещения в него очередной порции записей.

Стоимость операций со вторичной памятью

Природа устройств вторичной памяти (например, дисководов) такова, что время, необходимое для поиска блока и чтения его в основную память, достаточно велико в сравнении со временем, которое требуется для относительно простой обработки данных, содержащихся в этом блоке. Допустим, например, что у нас имеется блок из 1000 целых чисел на диске, вращающемся со скоростью 1000 об/мин. Время, которое требуется для позиционирования считывающей головки над дорожкой, содержащей этот блок (так называемое *время установки головок),* плюс время, затрачиваемое на ожидание, пока требуемый блок сделает оборот и окажется под головкой *(время ожидания*), может в среднем составлять 100 миллисекунд. Процесс записи блока в определенное место во вторичной памяти занимает примерно столько же времени. Однако за те же 100 миллисекунд машина, как правило, успевает выполнить 100 000 команд. Этого времени более чем достаточно, чтобы выполнить простую обработку тысячи целых чисел, когда они находятся в основной памяти (например, их суммирование или нахождение среди них наибольшего числа). Этого времени может даже хватить для выполнения быстрой сортировки целых чисел.

Оценивая время работы алгоритмов, в которых используются данные, хранящиеся в виде файлов, нам придется, таким образом, в первую очередь учитывать количество обращений к блокам, т.е. сколько раз мы считываем в основную память или записываем блок во вторичную память. Такая операция называется *доступом (или обращением) к блоку.* Предполагается, что размер блока фиксирован в операционной системе, поэтому у нас нет возможности ускорить работу алгоритма, увеличив размер блока и сократив тем самым количество обращений к блокам. Таким образом, мерой качества алгоритма, работающего с внешней памятью, является количество обращений к блокам. Изучение алгоритмов, работающих с внешней памятью, мы начнем с рассмотрения способов внешней сортировки.

11.2. Внешняя сортировка

Сортировка данных, организованных в виде файлов, или — в более общем случае — сортировка данных, хранящихся во вторичной памяти, называется *внешней сортировкой*. Приступая к изучению внешней сортировки, сделаем предположение, что данные хранятся в Pascal-файле. Мы покажем, как алгоритм *сортировки слиянием* позволяет отсортировать файл с n записями всего лишь за $O(\log n)$ проходов через файл; этот показатель намного лучше, чем $O(n)$ проходов, которые требовались алгоритмам, изучавшимся в главе 8. Затем мы рассмотрим, как использовать определенные возможности операционной системы по управлению чтением и записью блоков, что может ускорить сортировку за счет сокращения времени "бездействия" компьютера (периоды ожидания, пока блок будет прочитан в основную память или записан из основной памяти во внешнюю).

Сортировка слиянием

Главная идея, которая лежит в основе сортировки слиянием, заключается в том, что мы организуем файл в виде постепенно увеличивающихся *серий*, т.е. последовательностей записей $r_1, \dots, r_k$, где ключ r_i не больше, чем ключ r_{i+1}, $1 < i < k$.[1] Мы говорим, что файл, состоящий из $r_1, \dots, r_m$ записей, *делится на серии длиной* k, если для всех $i >$ О, таких, что $ki < m$ и $r_{k(i-1)+1}, r_{k(i-1)+2}, \dots, r_{ki}$ является последовательностью длиной k. Если m не делится нацело на k, т.е. $m = pk + q$, где $q < k$, тогда последовательность записей $r_{m-q+1}, r_{m-q+2}, — > r_m$, называемая *хвостом*, представляет собой серию длиной q. Например, последовательность целых чисел, показанная на рис. 11.1, организована сериями длиной 3. Обратите внимание, что хвост имеет длину, меньшую 3, однако и его записи тоже отсортированы.

7 15 29	8 11 13	16 22 31	5 12

Рис. 11.1. Файл с сериями длиной 3

Главное в сортировке файлов слиянием — начать с двух файлов, например f_1 и f_2, организованных в виде серий длиной k. Допустим, что (1) количества серий (включая хвосты) в Д. и f_2 отличаются не больше, чем на единицу; (2) по крайней мере один из файлов f_1 или f_2 имеет хвост; (3) файл с хвостом имеет не меньше серий, чем другой файл.

В этом случае можно использовать достаточно простой процесс чтения по одной серии из файлов f_1 и f_2, слияние этих серий и присоединения результирующей серии длиной $2k$ к одному из двух файлов g_1 и g_2, организованных в виде серий длиной $2k$. Переключаясь между g_1 и g_2, можно добиться того, что эти файлы будут не только организованы в виде серий длиной $2k$, но будут также удовлетворять перечисленным выше условиям (1) - (3). Чтобы выяснить, выполняются ли условия (2) и (3), достаточно убедиться в том, что хвост серий f_1 и f_2 слился с последней из созданных серий (или, возможно, уже был ею).

Итак, начинаем с разделения всех n записей на два файла f_1 и f_2 (желательно, чтобы записей в этих файлах было поровну). Можно считать, что любой файл состоит из серий длины 1. Затем мы можем объединить серии длины 1 и распределить их по файлам g_1 и g_2, организованным в виде серий длины 2. Мы делаем f_1 и fz пустыми и объединяем g_1 и g_2 в Д и f_2, которые затем можно организовать в виде серий длины 4. Затем мы объединяем Д и f_2, создавая g_1 и g_2, организованные в виде серий длиной 8, и т.д.

---

[1] Читатель, по-видимому, уже понял, что авторы здесь для простоты изложения материала одинаково обозначают записи и ключи этих записей. Но обратите внимание: в **листинге 11.1 предполагается**, что записи имеют отдельное поле *key* (ключ). — *Прим. ред.*

После выполнения i подобного рода проходов у нас получатся два файла, состоящие из серий длины 2^i. Если $2^i > n$, тогда один из этих двух файлов будет пустым, а другой будет содержать единственную серию длиной n, т.е. будет отсортирован. Так как $2^i > n$ при $i > \log n$, то нетрудно заметить, что в этом случае будет достаточно $[\log n] + 1$ проходов. Каждый проход требует чтения и записи двух файлов, длина каждого из них равна примерно $n/2$. Общее число блоков, прочитанных или записанных во время одного из проходов, составляет, таким образом, около $2n/b$, где b — количество записей, умещающихся в одном блоке. Следовательно, количество операций чтения и записи блоков для всего процесса сортировки равняется $O((n \log n)/b)$, или, говоря по-другому, количество операций чтения и записи примерно такое же, какое требуется при выполнении $O(\log n)$ проходов по данным, хранящимся в единственном файле. Этот показатель является существенным улучшением в сравнении с $O(n)$ проходами, которые требуются многим из алгоритмов сортировки, изучавшихся в главе 8.

В листинге 11.1 показан код программы сортировки слиянием на языке Pascal. Мы считываем два файла, организованных в виде серий длины k, и записываем два файла, организованных в виде серий длины $2k$. Предлагаем читателям, воспользовавшись изложенными выше идеями, самостоятельно разработать алгоритм сортировки файла, состоящего из n записей. В этом алгоритме должна $\log n$ раз использоваться процедура *merge* (слияние), представленная в листинге 11.1.

Листинг 11.1. Сортировка слиянием

```
procedure merge ( k: integer; { длина входной серии }
        f1, f2, g1, g2: file of recordtype);
var
    outswitch: boolean;
        { равна true, если идет запись в g1 и false, если в g2 }
    winner: integer;
        { номер файла с меньшим ключом в текущей записи }
    used: array[1..2] of integer;
        { used[j] сообщает, сколько записей прочитано
               к настоящему времени из текущей серии файла fj }
    fin: array[1..2] of boolean;
        { fin[j]=true, если уже закончена серия из файла fj:
            либо прочитано k записей, либо достигнут конец файла fj}
    current: array[1..2] of recordtype;
        { текущие записи из двух файлов }

procedure getrecord ( i: integer);
    { Перемещение по файлу fi, не выходя за конец файла или
        конец серии. Устанавливается fin[i]=true, если достигнут
        конец серии или файла }
    begin
        used[i]:= used[i] + 1;
        if (used[i]= k) or
            (i = 1) and eof(f1) or
            (i = 2) and eof(f2) then fin[i]:= true
        else if i = 1 then read(f1, current[1])
        else read(f2, current[2])
    end; { getrecord }

begin { merge }
    outswitch:= true; {первая объединенная серия записывается в g1}
    rewrite(g1); rewrite(g2);
    reset(f1); reset(f2);
```

```
while not eof(f1) or not eof(f2) do begin { слияние двух файлов }
        { инициализация }
        used[1]:= 0; used[2]:= 0;
        fin[1]:= false; fin[2]:= false;
        getrecord(1); getrecord(2);
        while not fin[1] or not fin[2] do begin { слияние серий }
            { вычисление переменной winner (победитель) }
            if fin[1] then winner:= 2
                {f2 "побеждает" по умолчанию: серия из f1 исчерпалась}
            else if fin[2] then winner:= 1
                {f1 "побеждает" по умолчанию}
            else {не исчерпалась ни одна из серий}
                if current[1].key < current[2].key then winner:= 1
                else winner:= 2;
            if outswitch then write(gl, current[winner])
            else write(g2, current[winner]);
            getrecord(winner)
        end;
        { закончено слияние двух серий, далее надо "переключить"
            выходной файл и повторить процедуру }
        outswitch:= not outswitch
    end
end; { merge }
```

Обратите внимание, что процедура *merge*, показанная в листинге 11.1, вовсе не требует, чтобы отдельная серия полностью находилась в памяти: она считывает и записывает последовательно запись за записью. Именно нежелание хранить целые серии в основной памяти заставляет нас использовать два входных файла. В противном случае можно было бы читать по две серии из одного файла одновременно.

Таблица 11.1. Сортировка слиянием

28	3	93	10	54	65	30	90	10	69	8	22
31	5	96	40	85	9	39	13	8	77	10	

а) исходные файлы

28 31	93 96	54 85	30 39	8 10	8 10
3 5	10 40	9 65	13 90	69 77	22

б) серии длиной 2

3 5 28 31	9 54 65 85	8 10 69 77
10 40 93 96	13 30 39 90	8 10 22

в) серии длиной 4

3 5 10 28 31 40 93 96	8 8 10 10 22 69 77
9 13 30 39 54 65 85 90	

г) серии длиной 8

3 5 9 10 13 28 30 31 39 40 54 65 85 90 93 96
8 8 10 10 22 69 77

д) серии длиной 16

3 5 8 8 9 10 10 10 13 22 28 30 31 39 40 54 65 69 77 85 90 93 96

с) серии длиной 32

Пример 11.1. Допустим, есть список из 23 чисел, поделенный на два файла (табл. 11.1,а). Мы начинаем с объединения серий длины 1, создавая два файла, представленных в табл. 11.1,б. Например, первыми сериями длины 1 являются 28 и 31; мы объединяем их, выбрав сначала 28, а затем 31. Следующие две серии единичной длины, 3 и 5, объединяются, образовывая серию длиной 2, и эта серия помещается во второй файл, показанный в табл. 11.1,б. Эти серии разделены в табл. 11.1,б вертикальными линиями, которые, разумеется, не являются частью файла. Обратите внимание, что второй файл в табл. 11.1,б имеет хвост единичной длины (запись 22), в то время как у первого файла хвоста нет.

Мы переходим от табл. 11.1,б к табл. 11.1,в, объединяя серии длины 2. Например, две такие серии (28, 31) и (3, 5) объединяются, образовывая серию (3, 5, 28, 31), показанную в табл. 11.1,в. К моменту, когда мы перейдем к сериям длиной 16 (см. табл. 11.1,д), один файл будет содержать одну лолную серию, а другой — только хвост длиной 7. На последней стадии, когда файлы должны быть организованы в виде серий длиной 32, у нас будет один файл, содержащий только хвост (длиной 23), и пустой второй файл. Единственная серия длиной 32 будет, конечно же, искомой отсортированной последовательностью чисел. □

Ускорение сортировки слиянием

Мы продемонстрировали пример процедуры сортировки слиянием, которая начинается с серий длины 1. Мы сэкономили бы немало времени, если бы начали эту процедуру с прохода, который считывает в основную память группы из k записей (при соответствующем k), сортирует их (например, с помощью процедуры быстрой сортировки) и записывает во внешнюю память в виде серии длиной k.

Если, например, у нас есть миллион записей, нам потребуется 20 проходов по этим данным, чтобы выполнить сортировку, начиная с серий длиной 1. Если, однако, у нас есть возможность одновременно поместить в основную память 10 000 записей, то мы сможем за один проход прочитать 100 групп из 10 000 записей, отсортировать каждую группу и получить таким образом 100 серий длиной 10 000, поделенных поровну между двумя файлами. Таким образом, всего семь проходов и слияний потребовалось бы для сортировки файла, содержащего не более $10\,000 \times 2^7 = 1\,280\,000$ записей.

Минимизация полного времени выполнения

В современных компьютерных системах с разделением времени пользователю обычно не приходится платить за время, в течение которого его программа ожидает считывания блоков данных из файла (операция, характерная для процесса сортировки слиянием). Между тем, полное время выполнения сортировки превышает (зачастую значительно) время обработки данных, находящихся в основной памяти. Если же нам приходится сортировать действительно большие файлы, время обработки которых измеряется часами, полное время становится критической величиной, даже если мы не платим за него из собственного кармана, и проблема минимизации полного времени процесса сортировки слиянием выходит на первый план.

Как уже указывалось, время, необходимое для считывания данных с магнитного диска или магнитной ленты, как правило, существенно превышает время, затрачиваемое на выполнение простых вычислений с этими данными (например, слияния списков). Таким образом, можно предположить, что при наличии лишь одного канала, по которому происходит обмен данными с основной памятью, именно этот канал и станет тем "узким местом", которое будет тормозить работу системы в целом. Этот канал обмена данными все время будет занят, и полное время работы системы будет практически равно времени, затрачиваемому на обмен данными с основной памятью, т.е. все вычисления будут выполняться практически мгновенно после того, как появятся соответствующие данные, и одновременно с тем, пока будет считываться или записываться следующая порция данных.

Даже в условиях такой относительно простой вычислительной среды следует позаботиться о минимизации затрат времени. Чтобы увидеть, что может произойти, если пренебречь этим требованием о минимизации временных затрат, допустим, что мы выполняем попеременное поблочное считывание двух входных файлов f_1 и f_2. Файлы организованы в виде серий определенной длины, намного превышающей размер блока, поэтому, чтобы объединить две такие серии, нам нужно прочитать несколько блоков из каждого файла. Предположим, однако, что все записи в серии из файла f_1 предшествуют всем записям из файла f_2. В этом случае при попеременном считывании блоков все блоки из файла f_2 должны оставаться в основной памяти. Основной памяти может не хватить для всех этих блоков, но даже если и хватит, нам придется (после считывания всех блоков серии) подождать, пока не будет скопирована и записана вся серия из файла f_2.

Чтобы избежать подобных проблем, мы рассматриваем ключи последних записей в последних блоках, считанных из Д и f_2, например ключи ftj и k_2 соответственно. Если какая-либо из серий исчерпалась, мы, естественно, считываем следующую серию из другого файла. Если серия не исчерпалась, мы считываем блок из файла Д, если, конечно, fei < k_2 (в противном случае считываем блок из f_2). То есть, мы определяем, у какой из двух серий будут первой выбраны все ее записи, находящиеся в данный момент в основной памяти, и в первую очередь пополняем запас записей именно для этой серии. Если выбор записей происходит быстрее, чем считывание, мы знаем, что когда будет считан последний блок этих двух серий, для последующего слияния не может остаться больше двух полных блоков записей; возможно, эти записи будут распределены по трем (максимум!) блокам.

Многоканальное слияние

ЕСЛИ "узким местом" является обмен данными между основной и вторичной памятью, возможно, удалось бы сэкономить время за счет увеличения числа каналов обмена данными. Допустим, *что* в нашей системе имеется $2m$ дисководов, каждый из которых имеет собственный канал доступа к основной памяти. Мы могли бы разместить на m дисководах m файлов (Д, f_2, ..., f_m), организованных в виде серий длины k. Тогда можно прочитать m серий, по одной из каждого файла, и объединить их в одну серию длиной mk. Эта серия помещается в один из m выходных файлов ($g_1, g_2, ..., g_m$), каждый из которых получает по очереди ту или иную серию.

Процесс слияния в основной памяти можно выполнить за $O(\log m)$ шагов на одну запись, если мы организуем *m записей-кандидатов*, т.е. наименьших на данный момент невыбранных записей из каждого файла, в виде частично упорядоченного дерева или другой структуры данных, которая поддерживает операторы INSERT и DELETEMIN, выполняемые над очередями с приоритетами за время порядка $O(\log n)$. Чтобы выбрать из очереди с приоритетами запись с наименьшим ключом, надо выполнить оператор DELETEMIN, а затем вставить (оператор INSERT) в очередь с приоритетами следующую запись из файла-победителя в качестве замены выбранной записи.

Если у нас имеется n записей, а длина серий после каждого прохода умножается на m, тогда после i проходов серии будут иметь длину m^i. Если $m^i > n$, т.е. после $i = \log_m n$ проходов, весь список будет отсортирован. Так как $\log_m n = \log_2 n / \log_2 m$, то "коэффициент экономии" (по количеству считываний каждой записи) составляет $\log_2 m$. Более того, если m — количество дисководов, используемых для входных файлов, и m дисководов используются для вывода, мы можем обрабатывать данные в m раз быстрее, чем при наличии лишь одного дисковода для ввода и одного дисковода для вывода, и в $2m$ раз быстрее, чем при наличии лишь одного дисковода для ввода и вывода (входные и выходные файлы хранятся на одном диске). К сожалению, бесконечное увеличение m не приводит к ускорению обработки по "закону коэффициента $\log m$". Причина заключается в том, что при достаточно больших значениях m время, необходимое для слияния в основной памяти (которое растет фактически пропорционально $\log m$), превосходит время, требующее-

ся для считывания или записи данных. Начиная с этого момента дальнейшее увеличение m ведет, по сути, к увеличению полного времени обработки данных, поскольку "узким местом" системы становятся вычисления в основной памяти.

Многофазная сортировка

Многоканальную (**m-канальную**) сортировку слиянием можно выполнить с помощью лишь $m + 1$ файлов (в отличие от описанной выше **$2m$-файловой** стратегии). При этом выполняется ряд проходов с объединением серий из m файлов в более длинные серии в $(m + 1)$-м файле. Вот последовательные шаги такой процедуры.

1. В течение одного прохода, когда серии от каждого из m файлов объединяются в серии $(m + 1)$-го файла, нет нужды использовать все серии от каждого из m входных файлов. Когда какой-либо из файлов становится **выходным,** он заполняется сериями определенной длины, причем количество этих серий равно минимальному количеству серий, находящихся в сливаемых файлах.
2. В результате каждого прохода получаются файлы разной длины. Поскольку каждый из файлов, загруженных сериями в результате предшествующих m проходов, вносит свой вклад в серии текущего прохода, длина всех серий на определенном проходе представляет собой сумму длин серий, созданных за предшествующие m проходов. (Если выполнено менее m проходов, можно считать, что гипотетические проходы, выполненные до первого прохода, создавали серии длины 1.)[1]

Подобный процесс сортировки слиянием называется *многофазной сортировкой.* Точный подсчет требуемого количества проходов как функции от m (количества файлов) и n (количество записей), а также нахождения оптимального начального распределения серий между m файлами оставлен для упражнений. Однако мы приведем здесь один пример общего характера.

Пример 11.2. Если $m = 2$, мы начинаем с двух файлов f_1 и f_2, организованных в виде серий длины 1. Записи из f_1 и f_2 объединяются, образуя серии длины 2 в третьем файле f_3. Выполняется слияние серий до полного опустошения файла f_1 (здесь для определенности предполагаем, что в файле f_1 меньше записей, чем в файле f_2). Затем объединяем оставшиеся серии длины 1 из f_2 с таким же количеством серий длины 2 из f_3. В результате получаются серии длины 3, которые помещаются в файл f_1. Затем объединяем серии длины 2 из f_3 с сериями длины 3 из f_1. Эти серии длиной 5 помещаются в файл /2. который был исчерпан во время предыдущего прохода.

Последовательность длин серий 1, 1, 2, 3, 5, 8, 13, 21, ... представляет собой последовательность *чисел Фибоначчи.* Эта последовательность удовлетворяет рекуррентному соотношению $F_i = F_{i-1} + F_{i-2}$ для $i > 2$ с начальными значениями $F_0 = F_1 = 1$. Обратите внимание, что отношение последовательных чисел Фибоначчи F_{i-1}/F_i приближается к "золотому соотношению" $(\sqrt{5}+1)/2 = 1.618...$ по мере увеличения i.

Оказывается, чтобы сохранить нужный ход процесса сортировки (пока не будет отсортирован весь список), начальные количества записей в f_1 и f_2 должны представлять собой два последовательных числа Фибоначчи. Например, в табл. 11.2 показано, что произойдет, если мы начнем наш процесс с $n = 34$ записями (34 — число Фибоначчи *FS),* распределенными следующим образом: 13 записей в файле f_1 и 21 запись в файле f_2 (13 и 21 представляют собой числа Фибоначчи F_6 и F_7, поэтому отношение F_7/F_6 равно 1.615, очень близко к 1.618). Состояние файлов в табл. 11.2 показано как *a(Ъ),* что означает *a* серий длиной *Ь.* П

---

[1] Существуют и другие зависимости между длинами серий на разных этапах. Например, длина серий в файле слияния равна сумме длин серий всех файлов предыдущего этапа. — *Прим. ред.*

Таблица 11.2. Пример **многофазной** сортировки

После прохода	f_1	f_2	f_3
Вначале	13(1)	21(1)	Пустой
1	Пустой	8(1)	13(2)
2	8(3)	Пустой	5(2)
3	3(3)	5(5)	Пустой
4	Пустой	2(5)	3(8)
5	2(13)	Пустой	1(8)
6	1(13)	1(21)	Пустой
7	Пустой	Пустой	1(34)

Когда скорость ввода-вывода не является „узким местом"

Когда "узким местом" является считывание файлов, необходимо очень тщательно выбирать блок, который должен считываться следующим. Как мы уже указывали, нужно избегать ситуаций, когда требуется запоминать много блоков одной серии, поскольку в этой серии наверняка имеются записи с большими значениями ключей, которые будут выбраны только после большинства (или всех) записей другой серии. Чтобы избежать этой ситуации, нужно быстро определить, какая серия первой исчерпает те свои записи, которые в данный момент находятся в основной памяти (эту оценку можно сделать, сравнив последние считанные записи из каждого файла).

Если время, необходимое для считывания данных в основную память, сопоставимо со временем, которое занимает обработка этих данных (или даже меньше его), тщательный выбор входного файла, из которого будет считываться блок, становится еще более важной задачей, поскольку иначе трудно сформировать резерв записей в основной памяти.

Рассмотрим случай, когда "узким местом" является слияние, а не считывание или запись данных. Это может произойти по следующим причинам.

1. Как мы уже видели, если в нашем распоряжении есть много дисководов или накопителей на магнитной ленте, ввод-вывод можно ускорить настолько, что время для выполнения слияния превысит время ввода-вывода.
2. Может стать экономически выгодным применение более быстродействующих каналов обмена данными.

Поэтому имеет смысл подробнее рассмотреть проблему, с которой можно столкнуться в случае, когда "узким местом" в процессе сортировки слиянием данных, хранящихся во вторичной памяти, становится их объединение. Сделаем следующие предположения.

1. Мы объединяем серии, размеры которых намного превышают размеры блоков.
2. Существуют два входных и два выходных файла. Входные файлы хранятся на одном внешнем диске (или каком-то другом устройстве, подключенном к основной памяти одним каналом), а выходные файлы — на другом подобном устройстве с одним каналом.
3. Время считывания, записи и выбора для заполнения блока записей с наименьшими ключами среди двух серий, находящихся в данный момент в основной памяти, одинаково.

С учетом этих предположений рассмотрим класс стратегий слияния, которые предусматривают выделение в основной памяти нескольких входных буферов (место для хранения блока). В каждый момент времени какой-то из этих буферов будет содержать невыделенные для слияния записи из двух входных серий, причем одна из них

будет находиться в состоянии считывания из входного файла. Два других буфера будут содержать выходные записи, т.е. выделенные записи в надлежащим образом объединенной последовательности. В каждый момент времени один из этих буферов находится в состоянии записи в один из выходных файлов, а другой заполняется записями, выбранными из входных буферов.

Выполняются (возможно, одновременно) следующие действия.

1. Считывание входного блока во входной буфер.
2. Заполнение одного из выходных буферов выбранными записями, т.е. записями с наименьшими ключами среди тех, которые в настоящий момент находятся во входном буфере.
3. Запись данных другого выходного буфера в один из двух формируемых выходных файлов.

В соответствии с нашими предположениями, эти действия занимают одинаковое время. Для обеспечения максимальной эффективности их следует выполнять параллельно. Это можно делать, если выбор записей с наименьшими ключами не включает записи, считываемые в данный момент.[1] Следовательно, мы должны разработать такую стратегию выбора буферов для считывания, чтобы в начале каждого этапа (состоящего из описанных действий) *b* невыбранных записей с наименьшими ключами уже находились во входных буферах (*Ъ* — количество записей, которые заполняют блок или буфер).

Условия, при которых слияние можно выполнять параллельно со считыванием, достаточно просты. Допустим, k_1 и k_2 — наибольшие ключи среди невыбранных записей в основной памяти из первой и второй серий соответственно. В таком случае в основной памяти должно быть по крайней мере *Ъ* невыбранных записей, ключи которых не превосходят $\min(k_1, k_2)$. Сначала мы покажем, как можно выполнить слияние с шестью буферами (по три на каждый файл), а затем покажем, что будет достаточно четырех буферов, если они будут использоваться совместно для двух файлов.

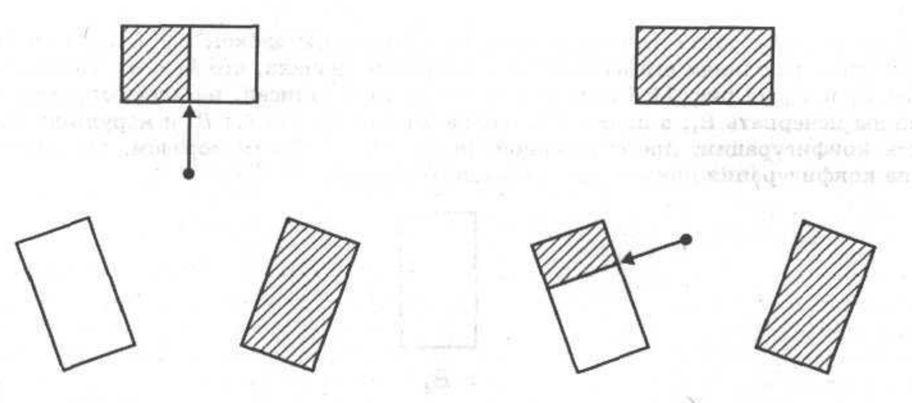
Схема с шестью входными буферами

Схема работы с шестью входными буферами представлена на рис. 11.2 (два выходных буфера здесь не показаны). Для каждого файла предусмотрены три буфера. Каждый буфер рассчитан на *Ъ* записей. Заштрихованная область представляет имеющиеся записи, ключи расположены по окружности (по часовой стрелке) в возрастающем порядке. В любой момент времени общее количество невыбранных записей равняется 46 (если только не рассматриваются записи, оставшиеся от объединяемых серий). Поначалу мы считываем в буферы первые два блока из каждой серии.[2] Поскольку у нас всегда имеется 46 записей, а из одного файла может быть не более 36 записей, мы знаем, что имеется по крайней мере *Ь* записей из каждого файла. Если k_1 и k_2 — наибольшие имеющиеся ключи в двух данных сериях, должно быть 6 записей с ключами, не большими, чем k_1, и 6 записей с ключами, не большими, чем ft_2. Таким образом, имеются 6 записей с ключами, не большими, чем $\min(k_1, ft_2)$.

---

[1] Напрашивается следующее предположение: если действия 1 и 2 занимают одинаковое время, значит, операция выбора никогда не пересечется со считыванием. Если целый блок еще не прочитан, можно было бы выбирать среди первых записей этого блока — тех, которые содержат меньшие ключи. Однако природа считывания с дисков такова, что должно пройти достаточно длительное время, прежде чем будет найден нужный блок и хотя бы что-нибудь считано из него. Таким образом, единственное, о чем можно говорить с уверенностью, это то, что на определенной стадии считывания никакие данные из считываемого блока не доступны для выбора.

[2] Если это не первые серии из каждого файла, тогда такую инициализацию можно сделать после того, как будут считаны предыдущие серии и выполнится объединение последних 46 записей из этих серий.



Буферы для файла 1 Буферы для файла 2

Рис. 11.2. Схема слияния с шестью входными буферами

Вопрос о том, какой файл считывать следующим, тривиален. Как **правило,** поскольку два буфера будут заполнены частично (как показано на рис. 11.2), в нашем распоряжении будет только один пустой буфер, который и следует заполнять. Если получается, что каждая серия имеет два полностью заполненных и один пустой буфер, используйте для заполнения любой из двух пустых буферов. Обратите внимание: наше утверждение о том, что мы не можем исчерпать серию (имеются Ъ записей с ключами, не большими, чем $\min(k_1, k_2)$), опирается исключительно на факт наличия 46 записей.

Стрелки на рис. 11.2 изображают указатели на первые (с наименьшими ключами) имеющиеся записи из двух данных серий. В языке Pascal можно представить такой указатель в виде двух целых чисел. Первый, в диапазоне 1-3, представляет "указываемый" буфер, а второй, в диапазоне $1-b$, — запись в этом буфере. Как альтернативный вариант мы могли бы реализовать эти буферы в виде первой, средней и последней трети одного массива и использовать одно целое число в диапазоне 1-36. При использовании других языков, в которых указатели могут указывать на элементы массивов, следует предпочесть указатель типа Trecordtype.

Схема с четырьмя буферами

На рис. 11.3 представлена схема с четырьмя буферами. В начале каждого этапа у нас имеется 26 записей. Два входных буфера назначаются одному из файлов (B_1 и B_2 на рис. 11.3 назначены файлу 1). Один из этих буферов будет заполнен частично (в крайнем случае он будет пустым), а другой — полностью. Третий буфер назначается другому файлу, например B_3 на рис. 11.3 назначен файлу 2. Он заполнен частично (в крайнем случае он будет заполнен полностью). Четвертый буфер не назначается ни одному из файлов. На данной стадии он заполняется из одного из этих файлов.

Мы, конечно, сохраняем возможность выполнения слияния параллельно со считыванием: по крайней мере Ъ записей из тех, которые показаны на рис. 11.3, должны иметь ключи, не превышающие $\min(k_1, k_2)$, где k_1 и k_2 — ключи последних имеющихся записей из двух данных файлов. Конфигурацию буферов, которая позволяет выполнять параллельную работу по слиянию и считыванию, назовем *безопасной*. Поначалу считывается один блок из каждого файла (крайний случай, когда буфер B_1 пустой, а буфер B_3 целиком заполнен); в результате начальная конфигурация оказывается безопасной. Мы должны (в предположении, что ситуация, представленная на рис. 11.3, соответствует безопасной конфигурации) показать, что конфигурация будет безопасной и после завершения следующего этапа.

Если $k_1 < k_2$, тогда надо заполнить B_4 следующим блоком из файла 1; в противном случае заполняем его из файла 2. Допустим сначала, что ft, < k_2. Поскольку буферы B_1 и B_3 на рис. 11.3 содержат в точности Ъ записей, на следующем этапе мы должны исчерпать B_1; в противном случае мы исчерпали бы B_3 и нарушили безопасность конфигурации, представленной на рис. 11.3. Таким образом, по завершении этапа конфигурация примет вид, показанный на рис. **11.4,а**.

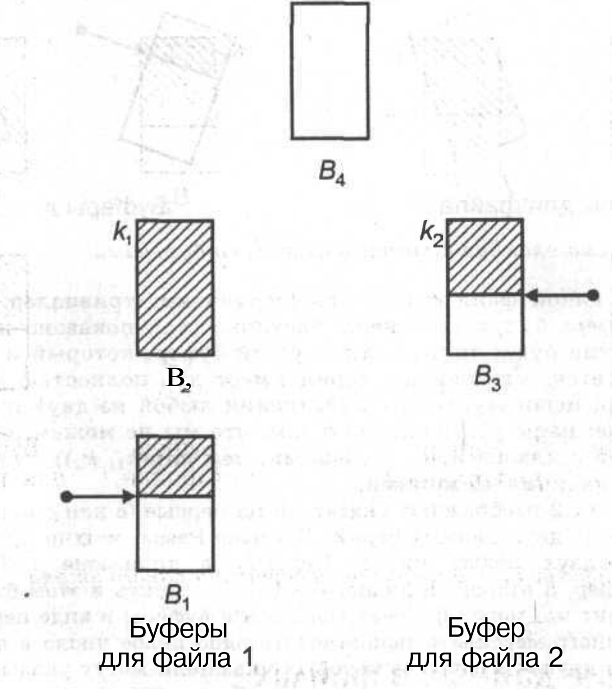


Рис. 11.3. Схема слияния с четырьмя буферами

Чтобы убедиться в том, что конфигурация на рис. **11.4,а** действительно безопасна, рассмотрим два случая. Во-первых, если k_3 (последний ключ во вновь прочитанном блоке B_4) оказывается меньше k_2, тогда при целиком заполненном блоке B_4 можно быть уверенным в наличии Ъ записей, не превышающих $\min(k_1, k_2)$, поэтому соответствующая конфигурация является безопасной. Если $k_2 \le k_3$, тогда в силу предположения, что $k_1 < k_2$ (в противном случае мы заполнили бы B_4 из файла 2), b записей в B_2 и B_3 имеют ключи, не превышающие $\min(k_2, k_3) - k_2$.

Теперь рассмотрим **случай**, когда fei > k_2 (см. рис. 11.3). Здесь необходимо считывать следующий блок из файла 2. На рис. **11.4,б** показана итоговая ситуация. Как и в случае $k_1 < k_2$, можно утверждать, что B_1 должен исчерпаться, вот почему на рис. **11.4,б** показано, что файл 1 имеет только буфер B_2. Доказательство того, что на рис. 11.4,6 показана безопасная конфигурация, ничем не отличается от доказательства подобного факта для рис. **11.4,а**.

Обратите внимание: как и в случае схемы с шестью входными буферами, мы не считываем файл после конца серии. Но если нет необходимости считывать блок из одной из имеющихся серий, мы может считать блок из следующей серии в том же файле. Таким образом, у нас появляется возможность считать один блок из каждой из следующих серий и приступить к слиянию серий сразу же после того, как будут выбраны последние записи предыдущей серии.

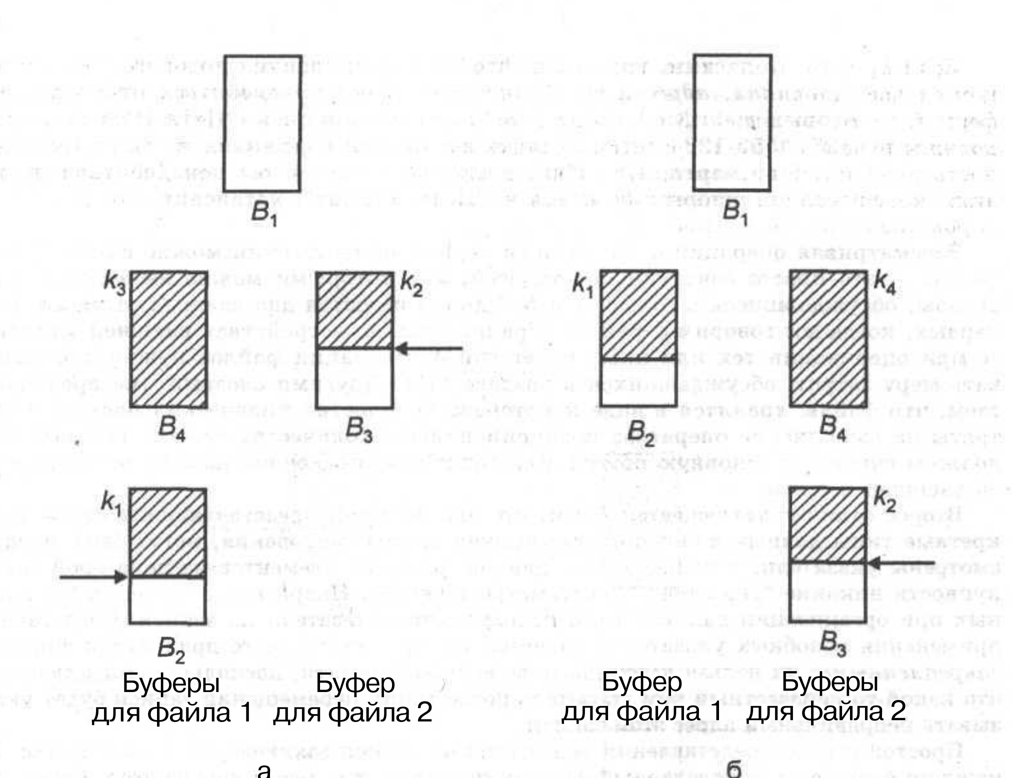


Рис. 11.4. Конфигурация буферов по завершении одного этапа

11.3. Хранение данных в файлах

В этом разделе мы рассмотрим структуры данных и алгоритмы для хранения и поиска информации в файлах, находящихся во внешней памяти. Файл мы будем рассматривать как последовательность записей, причем каждая запись состоит из одной и той же совокупности полей. Поля могут иметь либо *фиксированную длину* (заранее определенное количество байт), либо *переменную*. Файлы с записями фиксированной длины широко используются в системах управления базами данных для хранения данных со сложной структурой. Файлы с записями переменной длины, как правило, используются для хранения текстовой информации; в языке Pascal такие файлы не предусмотрены. В этом разделе будем иметь дело с полями фиксированной длины; рассмотренные методы работы после определенной (несложной) модификации могут использоваться для работы с записями переменной длины.

Мы рассмотрим следующие операторы для работы с файлами.

1. INSERT вставляет определенную запись в определенный файл.
2. DELETE удаляет из определенного файла все записи, содержащие указанные значения в указанных полях.
3. MODIFY изменяет все записи в определенном файле, задав указанные значения определенным полям в тех записях, которые содержат указанные значения в других полях.
4. RETRIEVE отыскивает все записи, содержащие указанные значения в указанных полях.

Пример 11.3. Допустим, например, что есть файл, записи которого состоят из трех полей: *фамилия, адрес* и *телефон.* Нам может понадобиться отыскать все записи, у которых *телефон* = "555-1234", вставить запись ("Петя Иванов", "ул. 8-го марта, 12", "555-1234") или удалить все записи с *фамилия* — "Петя Иванов" и *адрес* = "*ул.* 8-го марта, 12". Или, например, нам может понадобиться изменить все записи, у которых *фамилия* — "Петя Иванов", установив для поля *телефон* значение "555-1234". □

Рассматривая операции с файлами, в первом приближении можно считать, что файлы — это просто совокупности записей, над которыми можно выполнять операторы, обсуждавшиеся в главах 4 и 5. Однако имеются два важных отличия. Во-первых, когда мы говорим о файлах, хранящихся на устройствах внешней памяти, то при оценивании тех или иных стратегий организации файлов нужно использовать меру затрат, обсуждавшихся в разделе 11.1. Другими словами, мы предполагаем, что файлы хранятся в виде некоторого количества физических блоков, а затраты на выполнение оператора пропорциональны количеству блоков, которые мы должны считать в основную память или записать из основной памяти на устройство внешней памяти.

Второе отличие заключается в том, что для записей, представляющих собой конкретные типы данных в большинстве языков программирования, могут быть предусмотрены указатели, в то время как для абстрактных элементов из некоторой совокупности никакие "указатели" предусмотреть нельзя. Например, в системах баз данных при организации данных часто используются указатели на записи. Следствием применения подобных указателей является то, что записи часто приходится считать *закрепленными:* их нельзя перемещать во внешней памяти, поскольку не исключено, что какой-то неизвестный нам указатель после таких перемещений записи будет указывать неправильный адрес этой записи.

Простой способ представления указателей на записи заключается в следующем. У каждого блока есть определенный *физический адрес,* т.е. место начала этого блока на устройстве внешней памяти. Отслеживание физических адресов является задачей файловой системы. Одним из способов представления адресов записей является использование физического адреса блока, содержащего интересующую нас запись, со *смещением,* указывающим количество байт в блоке, предшествующих началу этой записи. Такие пары "физический адрес-смещение" можно хранить в полях типа "указатель на запись".

Простая организация данных

Простейшим (и наименее эффективным) способом реализации перечисленных выше операторов работы с файлами является использование таких примитивов чтения и записи файлов, которые встречаются, например, в языке Pascal. В случае использования подобной "организации" (которая на самом деле является дезорганизацией) записи могут храниться в любом порядке. Поиск записи с указанными значениями в определенных полях осуществляется путем полного просмотра файла и проверки каждой его записи на наличие в ней заданных значений. Вставку в файл можно выполнять путем присоединения соответствующей записи к концу файла.

В случае изменения записей необходимо просмотреть файл, проверить каждую запись и выяснить, соответствует ли она заданным условиям (значениям в указанных полях). Если соответствует, в запись вносятся требуемые изменения. Принцип действия операции удаления почти тот же, но когда мы находим запись, поля которой соответствуют значениям, заданным в операции удаления, мы должны найти способ удалить ее. Один из вариантов — сдвинуть все последовательные записи в своих блоках на одну позицию вперед, а первую запись в каждом последующем блоке переместить на последнюю позицию предыдущего блока данного файла. Однако такой подход не годится, если записи являются закрепленными, поскольку указатель на i-ю запись в файле после выполнения этой операции будет указывать на $(i + 1)$-ю запись.

Если записи являются закрепленными, нам следует воспользоваться каким-то другим подходом. Мы должны как-то помечать удаленные записи, но не должны смещать оставшиеся на место удаленных (и не должны вставлять на их место новые записи). Таким образом выполняется логическое удаление записи из файла, но ее место в файле остается незанятым. Это нужно для того, чтобы в случае появления указателя на удаленную запись мы могли, во-первых, понять, что указываемая запись уже удалена, и, во-вторых, предпринять соответствующие меры (например, присвоить этому указателю значение NIL, чтобы в следующий раз не тратить время на его анализ). Существуют два способа помечать удаленные записи.

1. Заменить запись на какое-то значение, которое никогда не может стать значением "настоящей" записи, и, встретив указатель на какую-либо запись, считать ее удаленной, если она содержит это значение.
2. Предусмотреть для каждой записи специальный *бит удаления;* этот бит содержит 1 в удаленных записях и 0 — в "настоящих" записях.

Ускорение операций с файлами

Очевидным недостатком последовательного файла является то, что операторы с такими файлами выполняются медленно. Выполнение каждой операции требует, чтобы мы прочитали весь файл, а после этого еще и выполнили перезапись некоторых блоков. К счастью, существуют такие способы организации файлов, которые позволяют нам обращаться к записи, считывая в основную память лишь небольшую часть файла.

Такие способы организации файлов предусматривают наличие у каждой записи файла так называемого *ключа,* т.е. совокупности полей, которая уникальным образом идентифицирует каждую запись. Например, в файле с полями *фамилия, адрес, телефон* поле *фамилия* само по себе может считаться ключом, т.е. мы можем предположить, что в таком файле не может одновременно быть двух записей с одинаковым значением поля *фамилия.* Поиск записи, когда заданы значения ее ключевых полей, является типичной операцией, на обеспечение максимальной эффективности которой ориентированы многие широко распространенные способы организации файлов.

Еще одним непременным атрибутом быстрого выполнения операций с файлами является возможность непосредственного доступа к блокам (в отличие от последовательного перебора всех блоков, содержащих файл). Многие структуры данных, которые мы используем для быстрого выполнения операций с файлами, используют указатели на сами блоки, которые представляют собой физические адреса этих блоков (о физических адресах блоков было сказано выше). К сожалению, на языке Pascal (и многих других языках программирования) невозможно писать программы, работающие с данными на уровне физических блоков и их адресов, — такие операции, как правило, выполняются с помощью команд файловой системы. Однако мы приведем краткое неформальное описание принципа действия операторов, в которых используется прямой доступ к блокам.

Хешированные файлы

Хеширование — широко распространенный метод обеспечения быстрого доступа к информации, хранящейся во вторичной памяти. Основная идея этого метода подобна открытому хешированию, которое мы обсуждали в разделе 4.7. Записи файла мы распределяем между так называемыми *сегментами,* каждый из которых состоит из связного списка одного или нескольких блоков внешней памяти. Такая организация подобна той, которая была представлена на рис. 4.3. Имеется таблица сегментов, содержащая B указателей, — по одному на каждый сегмент. Каждый указатель в таблице сегментов представляет собой физический адрес первого блока связного списка блоков для соответствующего сегмента.

Сегменты пронумерованы от 0 до $B-1$. Хеш-функция A отображает каждое значение ключа в одно из целых чисел от 0 до $B-1$. Если x — ключ, то $h(x)$ является номером сегмента, который содержит запись с ключом x (если такая запись вообще существует). Блоки, составляющие каждый сегмент, образуют связный список. Таким образом, заголовок i-го блока содержит указатель на физический адрес $(i + 1)$-го блока. Последний блок сегмента содержит в своем заголовке NIL-указатель.

Такой способ организации показан на рис. 11.5. Основное различие между рис. 11.5 и 4.3 заключается в том, что в данном случае элементы, хранящиеся в одном блоке сегмента, не требуется связывать друг с другом с помощью указателей — связывать между собой нужно только блоки.

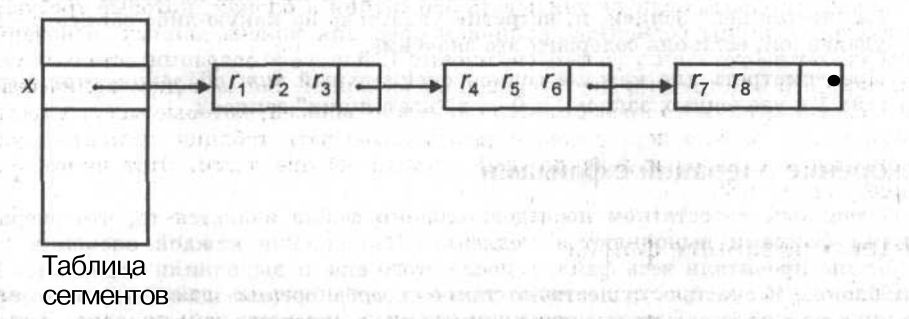


Рис. 11.5. Сегменты, состоящие из связанных блоков

Если размер таблицы сегментов невелик, ее можно хранить в основной памяти. В противном случае ее можно хранить последовательным способом в отдельных блоках. Если нужно найти запись с ключом x, вычисляется $h(x)$ и находится блок таблицы сегментов, содержащий указатель на первый блок сегмента $h(x)$. Затем последовательно считываются блоки сегмента $h(x)$, пока не обнаружится блок, который содержит запись с ключом x. Если исчерпаны все блоки в связном списке для сегмента $h(x)$, приходим к выводу, что x не является ключом ни одной из записей.

Такая структура оказывается вполне эффективной, если в выполняемом операторе указываются значения ключевых полей. Среднее количество обращений к блокам, требующееся для выполнения оператора, в котором указан ключ записи, приблизительно равняется среднему количеству блоков в сегменте, которое равно n/bk, если n — количество записей, блок содержит b записей, а k соответствует количеству сегментов. Таким образом, при такой организации данных операторы, использующие значения ключей, выполняются в среднем в k раз быстрее, чем в случае неорганизованного файла. К сожалению, ускорения операций, не основанных на использовании ключей, добиться не удается, поскольку при выполнении подобных операций нам приходится анализировать практически все содержимое сегментов. Единственным универсальным способом ускорения операций, не основанных на использовании ключей, по-видимому, является применение вторичных индексов, о которых мы поговорим в конце этого раздела.

Чтобы вставить запись с ключом, значение которого равняется x, нужно сначала проверить, нет ли в файле записи с таким значением ключа. Если такая запись есть, выдается сообщение об ошибке, поскольку мы предполагаем, что ключ уникальным образом идентифицирует каждую запись. Если записи с ключом x нет, мы вставляем новую запись в первый же блок цепочки для сегмента $h(x)$, в который эту запись удается вставить. Если запись не удается вставить ни в какой из существующих блоков сегмента $h(x)$, файловой системе выдается команда найти новый блок, в который будет помещена эта запись. Этот новый блок затем добавляется в конец цепочки блоков сегмента $h(x)$.

Чтобы удалить запись с ключом x, нужно сначала найти эту запись, а затем установить ее бит удаления. Еще одной возможной стратегией удаления (которой, впро-

чем, нельзя пользоваться, если мы имеем дело с закрепленными записями) является замена удаленной записи на последнюю запись в цепочке блоков сегмента $h(x)$. Если такое изъятие последней записи приводит к опустошению последнего блока в сегменте $h(x)$, этот пустой блок можно затем вернуть файловой системе для повторного использования.

Хорошо продуманная организация файлов с хешированным доступом требует лишь незначительного числа обращений к блокам при выполнении каждой операции с файлами. Если мы имеем дело с хорошей функцией хеширования, а количество сегментов приблизительно равно количеству записей в файле, деленному на количество записей, которые могут уместиться в одном блоке, тогда средний сегмент состоит из одного блока. Если не учитывать обращения к блокам, которые требуются для просмотра таблицы сегментов, типичная операция поиска данных, основанного на ключах, потребует лишь одного обращения к блоку, а операции вставки, удаления или изменения потребуют двух обращений к блокам. Если среднее количество записей в сегменте намного превосходит количество записей, которые могут уместиться в одном блоке, можно периодически реорганизовывать таблицу сегментов, удваивая количество сегментов и деля каждый сегмент на две части. Этот прием описан в конце раздела 4.8.

Индексированные файлы

Еще одним распространенным способом организации файла записей является поддержание файла в отсортированном (по значениям ключей) порядке. В этом случае файл можно было бы просматривать как обычный словарь или телефонный справочник, когда мы просматриваем лишь заглавные слова или фамилии на каждой странице. Чтобы облегчить процедуру поиска, можно создать второй файл, называемый *разреженным индексом*, который состоит из пар (x, b), где x — значение ключа, а b — физический адрес блока, в котором значение ключа первой записи равняется x. Этот разреженный индекс отсортирован по значениям ключей.

Пример 11.4. На рис. 11.6 показан файл, а также соответствующий ему файл разреженного индекса. Предполагается, что три записи основного файла (или три пары индексного файла) умещаются в один блок. Записи основного файла представлены только значениями ключей, которые в данном случае являются целочисленными величинами. □

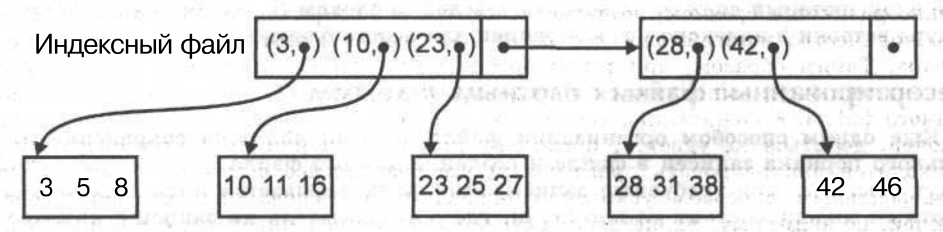


Рис. 11.6. Основной файл и его разреженный индекс

Чтобы отыскать запись с заданным ключом x, надо сначала просмотреть индексный файл, отыскивая в нем пару (x, b). В действительности отыскивается наибольшее z, такое, что $z \leq x$ и далее находится пара (z, b). В этом случае ключ x оказывается в блоке b (если такой ключ вообще присутствует в основном файле).

Разработано несколько стратегий просмотра индексного файла. Простейшей из них является *линейный поиск*. Индексный файл читается с самого начала, пока не встретится пара (x, b) или (y, b), причем $y > x$. В последнем случае у предыдущей пары (z, b') должно быть $z < x$, и если запись с ключом x действительно существует, она находится в блоке b'.

Линейный поиск годится лишь для небольших индексных файлов. Более эффективным методом является *двоичный поиск*. Допустим, что индексный файл хранится в блоках $b_1, b_2, \ldots, b_n$. Чтобы отыскать значение ключа *x*, берется средний блок fy$_n$/2] и *x* сравнивается со значением ключа *y* в первой паре данного блока. Если $x < y$, поиск повторяется в блоках $b_1, b_2, \ldots, b_{[n/2]-1}$. Если $x > y$, но *x* меньше, чем ключ блока fyn/2]+i» используется линейный поиск, чтобы проверить, совпадает ли x с первым компонентом индексной пары в блоке $b_{[n/2]}$. В противном случае повторяется поиск в блоках $b_{[n/2]+1}, b_{[n/2]+2}, — , b_n$. При использовании двоичного поиска нужно проверить лишь $[\log_2(n + 1)]$ блоков индексного файла.

Чтобы создать индексированный файл, записи сортируются по значениям их ключей, а затем распределяются по блокам в возрастающем порядке ключей. В каждый блок можно упаковать столько записей, сколько туда умещается. Однако в каждом блоке можно оставлять место для дополнительных записей, которые могут вставляться туда впоследствии. Преимущество такого подхода заключается в том, что вероятность переполнения блока, куда вставляются новые записи, в этом случае оказывается ниже (иначе надо будет обращаться к смежным блокам). После распределения записей по блокам создается индексный файл: просматривается по очереди каждый блок и находится первый ключ в каждом блоке. Подобно тому, как это сделано в основном файле, в блоках, содержащих индексный файл, можно оставить какое-то место для последующего "роста".

Допустим, есть отсортированный файл записей, хранящихся в блоках $B_1, B_2, \ldots, B_m$. Чтобы вставить в этот отсортированный файл новую запись, используем индексный файл, с помощью которого определим, какой блок B_i должен содержать новую запись. Если новая запись умещается в блок B_i, она туда помещается в правильной последовательности. Если новая запись становится первой записью в блоке B_i, тогда выполняется корректировка индексного файла.

Если новая запись не умещается в блок B_i, можно применить одну из нескольких стратегий. Простейшая из них заключается в том, чтобы перейти на блок B_{i+1} (который можно найти с помощью индексного файла) и узнать, можно ли последнюю запись B_i переместить в начало B_{i+1}. Если можно, последняя запись перемещается в B_{i+1}, а новую запись можно затем вставить на подходящее место в B_i. В этом случае, конечно, нужно скорректировать вход индексного файла для B_{i+1} (и, возможно, для B_i).

Если блок B_{i+1} также заполнен или если B_i является последним блоком ($i = m$), из файловой системы нужно получить новый блок. Новая запись вставляется в этот новый блок, который должен размещаться вслед за блоком B_i. Затем используется процедура вставки в индексном файле записи для нового блока,

Несортированные файлы с плотным индексом

Еще одним способом организации файла записей является сохранение произвольного порядка записей в файле и создание другого файла, с помощью которого будут отыскиваться требуемые записи; этот файл называется *плотным индексом*. Плотный индекс состоит из пар *(x, p)*, где *p* — указатель на запись с ключом *x* в основном файле. Эти пары отсортированы по значениям ключа. В результате структуру, подобную упоминавшемуся выше разреженному индексу (или B-дереву, речь о котором пойдет в следующем разделе), можно использовать для поиска ключей в плотном индексе.

При использовании такой организации плотный индекс служит для поиска в основном файле записи с заданным ключом. Если требуется вставить новую запись, отыскивается последний блок основного файла и туда вставляется новая запись. Если последний блок полностью заполнен, то надо получить новый блок из файловой системы. Одновременно вставляется указатель на соответствующую запись в файле плотного индекса. Чтобы удалить запись, в ней просто устанавливается бит удаления и удаляется соответствующий вход в плотном индексе (возможно, устанавливая и здесь бит удаления).

Вторичные индексы

В то время как хешированные и индексированные структуры ускоряют выполнение операций с файлами, основываясь главным образом на ключах, ни один из этих методов не помогает, когда операция связана с поиском записей, если заданы значения полей, не являющихся ключевыми. Если требуется найти записи с указанными значениями в некоторой совокупности полей $F_1, F_2, \ldots, F_k$, нам понадобится *вторичный индекс* для этих полей. Вторичный индекс — это файл, состоящий из пар (и, *p)*, где *v* представляет собой список значений, по одному для каждого из полей $F_1, F_2, \ldots, F_k$, а p — указатель на запись. В файле вторичного индекса может быть несколько пар с заданным v, и каждый сопутствующий указатель должен указывать на запись в основном файле, которая содержит *v* в качестве списка значений полей $F_1, F_2, \ldots, F_k$.

Чтобы отыскать записи, когда заданы значения полей $F_1, F_2, \ldots, F_k$, мы отыскиваем в файле вторичного индекса запись (или записи) с заданным списком значений. Сам файл вторичного индекса может быть организован любым из перечисленных выше способов организации файлов по значениям ключей. Таким образом, мы предполагаем, что *v* является ключом для пары *(v, p)*.

Например, организация файлов с помощью хеширования практически не зависит от того, уникальны ли ключи, — хотя, если бы оказалось очень много записей с одним и тем же значением "ключа", записи могли бы распределиться по сегментам очень неравномерно, и в результате хеширование не ускорило бы доступ к записям. Рассмотрим крайний случай, когда заданы только два значения для полей вторичного индекса. При этом все сегменты, за исключением двух, были бы пустыми, и таблица хеширования, независимо от имеющегося количества сегментов, ускоряла бы выполнение операций лишь в два раза (и то в лучшем случае). Аналогично, разреженный индекс не требует, чтобы ключи были уникальны, но если они действительно не будут уникальными, тогда в основном файле может оказаться два или больше блоков, содержащих одинаковые наименьшие значения "ключа", и когда нам потребуется найти записи с этим значением, необходимо будет просмотреть все такие блоки.

Если файл вторичного индекса организован по методу хеширования или разреженного индекса, может возникнуть желание сэкономить память, объединив все записи с одинаковыми значениями, т.е. пары (v, p_1), (v, p_2) ... (v, p_m) можно заменить на v, за которым следует список $p_1, p_2, \ldots, p_m$.

Может возникнуть вопрос, нельзя ли оптимизировать время выполнения операторов, создав вторичный индекс для каждого поля или даже для всех подмножеств полей. К сожалению, придется "платить" за каждый вторичный индекс, который мы хотим создать. Во-первых, для хранения вторичного индекса тоже требуется место на диске, а этот ресурс (объем внешней памяти) тоже, хоть и не всегда, может быть дефицитным. Во-вторых, каждый создаваемый вторичный индекс замедляет выполнение всех операций вставки и удаления. Дело в том, что когда вставляется запись, надо также вставить точку входа для этой записи в каждый вторичный индекс, чтобы эти вторичные индексы по-прежнему правильно представляли соответствующий файл. Обновление вторичного индекса означает, что потребуется выполнить не менее двух обращений к блокам, поскольку мы должны и прочитать, и записать блок. Однако таких обращений к блокам может оказаться значительно больше, поскольку еще нужно найти требуемый блок, а любой способ организации файла, который используется для вторичного индекса, потребует в среднем еще нескольких обращений, чтобы найти нужный блок. Все сказанное относится и к операции удаления записей. Отсюда следует вывод, что решение использовать вторичные индексы требует взвешенного подхода, поскольку надо определить, какие совокупности полей будут указываться в операциях, выполняемых с файлами, настолько то время, которое мы собираемся сэкономить за счет применения вторичных индексов, превосходит затраты на обновление этого индекса при выполнении каждой операции вставки и удаления.

11.4. Внешние деревья поиска

Древовидные структуры данных, которые обсуждались в главе 5 в качестве средства представления множеств, можно использовать для представления внешних файлов. В-дерево, являющееся обобщением 2-3 дерева (см. главу 5), особенно удачно подходит для представления внешней памяти и стало стандартным способом организации индексов в системах баз данных. В этом разделе мы познакомим читателей с базовыми методами поиска, вставки и удаления информации в В-деревьях.

Разветвленные деревья поиска

Обобщением дерева двоичного поиска является m-арное дерево, в котором каждый узел имеет не более m сыновей. Так же, как и для деревьев двоичного поиска, будем считать, что выполняется следующее условие: если n_1 и n_2 являются двумя сыновьями одного узла и n_1 находится слева от n_2, тогда все элементы, исходящие вниз от n_1, оказываются меньше элементов, исходящих вниз от n_2. Операторы MEMBER, INSERT и DELETE для m-арного дерева поиска реализуются путем естественного обобщения тех операций для деревьев двоичного поиска, которые обсуждались в разделе 5.1.

Однако в данном случае нас интересует проблема хранения записей в файлах, когда файлы хранятся в виде блоков внешней памяти. Правильным применением идеи разветвленного дерева является представление об узлах как о физических блоках. Внутренний узел содержит указатели на своих от сыновей, а также m - 1 ключевых значений, которые разделяют потомков этих сыновей. Листья также являются блоками; эти блоки содержат записи основного файла.

Если бы мы использовали дерево двоичного поиска из n узлов для представления файла, хранящегося во внешней памяти, то для поиска записи в таком файле нам потребовалось бы в среднем $\log_2 n$ обращений к блокам. Если бы вместо дерева двоичного поиска мы использовали для представления файла m-арное дерево поиска, то для поиска записи в таком файле нам потребовалось бы лишь $\log_m n$ обращений к блокам. В случае $n = 10^6$ дерево двоичного поиска потребовало бы примерно 20 обращений к блокам, тогда как 128-арное дерево поиска потребовало бы лишь 3 обращения к блокам.

Однако мы не можем сделать m сколь угодно большим, поскольку чем больше m, тем больше должен быть размер блока. Более того, считывание и обработка более крупного блока занимает больше времени, поэтому существует оптимальное значение m, которое позволяет минимизировать время, требующееся для просмотра внешнего m-арного дерева поиска. На практике значение, близкое к такому минимуму, получается для довольно широкого диапазона величин m. (См. упражнение 11.18.)

В-деревья

В-дерево — это особый вид сбалансированного от-арного дерева, который позволяет нам выполнять операции поиска, вставки и удаления записей из внешнего файла с гарантированной производительностью для самой неблагоприятной ситуации. Оно представляет собой обобщение 2-3 дерева, которое обсуждалось в разделе 5.4. С формальной точки зрения *В-дерево порядка m* представляет собой от-арное дерево поиска, характеризующееся следующими свойствами.

1. Корень либо является листом, либо имеет по крайней мере двух сыновей.
2. Каждый узел, за исключением корня и листьев, имеет от [т/2] до m сыновей.
3. Все пути от корня до любого листа имеют одинаковую длину.

Обратите внимание: каждое 2-3 дерево является В-деревом порядка 3, т.е. 3-арным. На рис. 11.7 показано В-дерево порядка 5, здесь предполагается, что в блоке листа умещается не более трех записей.

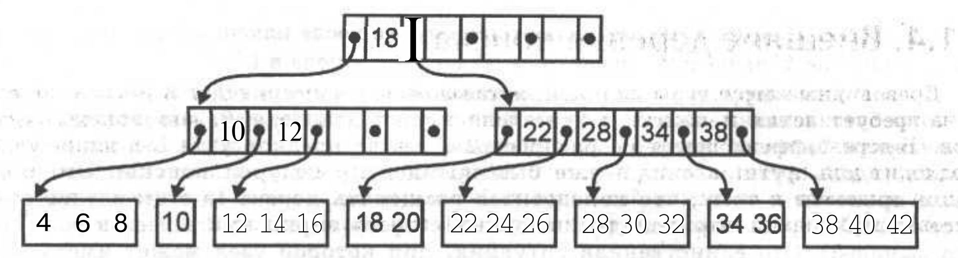


Рис. 11.7. В-дерево порядка 5

В-дерево можно рассматривать как иерархический индекс, каждый узел в котором занимает блок во внешней памяти. Корень В-дерева является индексом первого уровня. Каждый нелистовой узел на **В-дереве** имеет **форму** $(p_0, k_1, p_1, k_2, p_2, \ldots, k_n, p_n)$, где p_i является указателем на i-го сына, $0 < i < n$, а k_i — ключ, $1 < i < n$. Ключи в узле упорядочены, поэтому $k_1 < k_2 < \ldots < k_n$. Все ключи в поддереве, на которое указывает p_0, меньше, чем k_1. В случае $1 < i < n$ все ключи в поддереве, на которое указывает p_i, имеют значения, не меньшие, чем k_i, и меньшие, чем ft_{j+1}. Все ключи в поддереве, на которое указывает p_n, имеют значения, не меньшие, чем k_n.

Существует несколько способов организации листьев. В данном случае мы предполагаем, что записи основного файла хранятся только в листьях. Предполагается также, что каждый лист занимает один блок.

Поиск записей

Если требуется найти запись r со значением ключа x, нужно проследить путь от корня до листа, который содержит r, если эта запись вообще существует в файле. Мы проходим этот путь, последовательно считывая из внешней памяти в основную внутренние узлы (p_0, fe_{lt} p_1, ..., k_n, p_n) и вычисляя положение x относительно ключей k_1, k_2, ..., k_n. Если $k_i \le x < k_{i+1}$, тогда в основную память считывается узел, на который указывает p_i, и повторяется описанный процесс. Если $x < k_1$, для считывания в основную память используется указатель p_0; если $x \ge k_n$, тогда используется p_n. Когда в результате этого процесса мы попадаем на какой-либо **лист**, то пытаемся найти запись со значением ключа x. Если количество входов в узле невелико, то в этом узле можно использовать линейный поиск; в противном случае лучше воспользоваться двоичным поиском.

Вставка записей

Вставка в В-дерево подобна вставке в 2-3 дерево. Если требуется **вставить в** В-дерево запись r *со* значением ключа x, нужно сначала воспользоваться процедурой поиска, чтобы найти лист L, которому должна принадлежать запись r. Если в L есть место для этой записи, то она вставляется в L в требуемом порядке. В этом случае внесение каких-либо изменений в предков листа L не требуется.

Если же в блоке листа L нет места для записи r, у файловой системы запрашивается новый блок L' и перемещается последняя половина записей из L в L', вставляя r в требуемое место в L или L'.[1] Допустим, узел P является родителем узла L. P известен, поскольку процедура поиска отследила путь от корня к листу L через узел P. Теперь можно рекурсивно применить процедуру вставки, чтобы разместить в P ключ

[1] Эта стратегия представляет собой простейший из возможных вариантов реакции на ситуацию, когда необходимо "расщепить" блок. Некоторые другие варианты, обеспечивающие более высокую среднюю заполненность блоков за счет выполнения дополнительных **действий** при вставке каждой записи, упоминаются в упражнениях.

k' и указатель l' на L'; k' и l' вставляются сразу же после ключа и указателя для листа L. Значение ft'является наименьшим значением ключа в L'.

Если P уже имеет m указателей, вставка ft'и l' в P приведет к расщеплению P и потребует вставки ключа и указателя в узел родителя P. Эта вставка может произвести "эффект домино", распространяясь на предков узла L в направлении корня (вдоль пути, который уже был пройден процедурой поиска). Это может даже привести к тому, что понадобится расщепить корень (в этом случае создается новый корень, причем две половины старого корня выступают в роли двух его сыновей). Это единственная ситуация, при которой узел может иметь менее $m/2$ потомков.

Удаление записей

ЕСЛИ требуется удалить запись г со значением ключа x, нужно сначала найти лист L, содержащий запись r. Затем, если такая запись существует, она удаляется из L. Если г является первой записью в L, мы переходим после этого в узел P (родителя листа L), чтобы установить новое значение первого ключа для L. Но если L является первым сыном узла P, то первый ключ L не зафиксирован в P, а появляется в одном из предков P. Таким образом, надо распространить изменение в наименьшем значении ключа L в обратном направлении вдоль пути от L К корню.

Если блок листа L после удаления записи оказывается пустым, он отдается файловой системе.[1] После этого корректируются ключи и указатели в P, чтобы отразить факт удаления листа L. Если количество сыновей узла P оказывается теперь меньшим, чем $m/2$, проверяется узел P', расположенный в дереве на том же уровне непосредственно слева (или справа) от P. Если узел P' имеет по крайней мере $\lceil m/2 \rceil + 2$ сыновей, ключи и указатели распределяются поровну между P и P' так, чтобы оба эти узла имели по меньшей мере $\lceil m/2 \rceil$ потомков, сохраняя, разумеется, упорядочение записей. Затем надо изменить значения ключей для P и P' в родителе P и, если необходимо, рекурсивно распространить воздействие внесенного изменения на всех предков узла P, на которых это изменение отразилось.

Если у P' имеется ровно $\lceil m/2 \rceil$ сыновей, мы объединяем P и P' в один узел с $2\lceil m/2 \rceil$ - 1 сыновьями. Затем необходимо удалить ключ и указатель на P' из родителя для P'. Это удаление можно выполнить с помощью рекурсивного применения процедуры удаления.

Если "обратная волна" воздействия удаления докатывается до самого корня, возможно, нам придется объединить только двух сыновей корня. В этом случае новым корнем становится результирующий объединенный узел, а старый корень можно вернуть файловой системе. Высота В-дерева уменьшается при этом на единицу.

Пример 11.5. Рассмотрим В-дерево порядка 5, показанное на рис. 11.7. Вставка записи со значением ключа 23 порождает В-дерево, показанное на рис. 11.8. Чтобы вставить эту запись, надо расщепить блок, содержащий записи с ключами 22, 23, 24 и 26, поскольку предполагаем, что в один блок помещается не более трех записей. Два меньших остаются в этом блоке, а два больших помещаются в новый блок. Пару "указатель-ключ" для нового узла нужно вставить в родителя, который в таком случае расщепляется, поскольку не может содержать шесть указателей. Корень принимает пару "указатель-ключ" для нового узла, однако корень не расщепляется, поскольку он располагает достаточной емкостью.

Удаление записи с ключом 10 из В-дерева, показанного на рис. 11.8, приводит к В-дереву, показанному на рис. 11.9. В этом случае блок, содержащий запись с ключом 10, отбрасывается. У его родителя теперь оказывается только два

---

[1] Чтобы предотвратить возможное полное опустошение блоков листов, можно применять различные стратегии. В частности, ниже мы описываем схему предотвращения опустошения внутренних узлов более чем наполовину. Этот метод можно применить и к листьям.

сына, а у правого "брата" этого родителя имеется минимальное количество сыновей — три. Таким образом, мы объединяем родителя с его "братом" и получаем один узел с пятью сыновьями. □

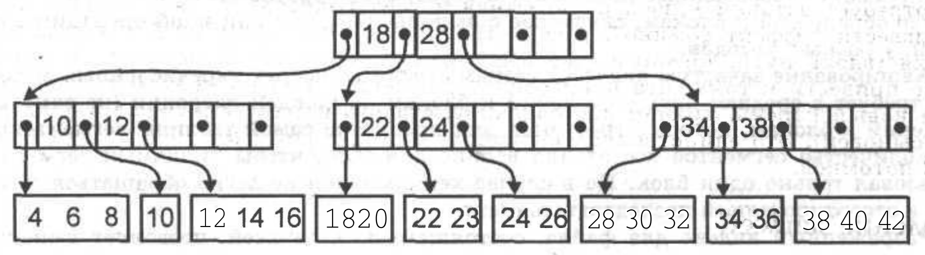


Рис. 11.8. В-дерево после вставки записи

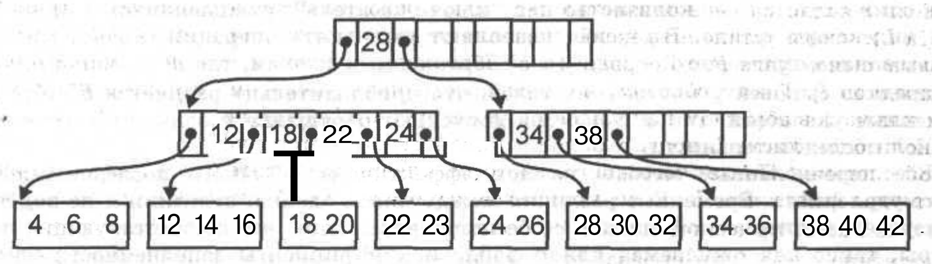


Рис. *11.9. В-дерево после удаления записи*

Время выполнения операций с В-деревом

Допустим, есть файл с n записями, организованный в виде В-дерева порядка m. Если каждый лист содержит в среднем b записей, тогда дерево содержит примерно $[n/b]$ листьев. Самые длинные пути в таком дереве образуются в том случае, если каждый внутренний узел имеет наименьшее возможное количество сыновей, т.е. $m/2$. В этом случае будет примерно $2[n/b]/m$ родителей листьев, $4[n/b]/m^2$ родителей родителей листьев и т.д.

Если вдоль пути от корня к листу встречается j узлов, в этом случае $2^{j-1}[n/b]/m^{j-1} > 1$; в противном случае на уровне корня окажется меньше одного узла. Таким образом, $[n/b] > (m/2)^{j-1}$, а $j < 1 + \log_{m/2}[n/b]$. Например, если $n = 10^6$, $b = 10$, а $m = 100$, то $j < 3{,}9$. Обратите внимание, что b является не максимальным количеством записей, которые мы можем поместить в блок, а средним или ожидаемым количеством. Однако, перераспределяя записи между соседними блоками каждый раз, когда один из них опустошается больше чем наполовину, мы можем гарантировать, что b будет равняться по крайней мере половине максимального значения. Обратите также внимание на высказанное нами выше предположение о том, что каждый внутренний узел имеет минимально возможное количество сыновей. На практике количество сыновей среднего внутреннего узла будет больше минимума, и приведенный выше анализ отражает, таким образом, самый "консервативный" случай.

В случае вставки или удаления записи для поиска нужного листа потребуется j обращений к блокам. Точное количество дополнительных обращений к блокам, необходимое для выполнения вставки или удаления записи и распространения воздействия этих операций по дереву, подсчитать весьма затруднительно. Чаще всего требуется перезаписать только один блок — лист, содержащий интересующую нас запись. Таким образом, в качестве ориентировочного количества обращений к блокам при выполнении вставки или удаления записи можно принять значение $2 + \log_{m/2}[n/b]$.

Сравнение методов

Итак, в качестве возможных методов организации внешних файлов мы обсудили хеширование, разреженные индексы и В-деревья. Интересно было бы сравнить количество обращений к блокам, связанное с выполнением той или иной операции с файлами, у разных методов.

Хеширование зачастую является самым быстрым из трех перечисленных методов; оно требует в среднем двух обращений к блокам по каждой операции (не считая обращений к блокам, которые требуются для просмотра самой таблицы сегментов), если количество сегментов достаточно велико для того, чтобы типичный сегмент использовал только один блок. Но в случае хеширования не легко обращаться к записям в отсортированной последовательности.

Разреженный индекс для файла, состоящего из л записей, позволяет выполнять операции с файлами, ограничиваясь использованием примерно $2 + \log(n/bb')$ обращений к блокам в случае двоичного поиска, где Ь — количество записей, помещающихся в один блок, а b' — количество пар "ключ-указатель", умещающихся в один блок для индексного файла. В-деревья позволяют выполнять операции с файлами с использованием примерно $2 + \log_{m/2}[n/b]$ обращений к блокам, где m — максимальное количество сыновей у внутренних узлов, что приблизительно равняется Ъ'. Как разреженные указатели, так и В-деревья допускают обращение к записям в отсортированной последовательности.

Все перечисленные методы намного эффективнее обычного последовательного просмотра файла. Временные различия между ними, однако, невелики и не поддаются точной аналитической оценке, особенно с учетом того, что соответствующие параметры, такие как ожидаемая длина файла и коэффициенты заполненности блоков, трудно прогнозировать заранее.

Похоже на то, что В-деревья приобретают все большую популярность как средство доступа к файлам в системах баз данных. Причина этой популярности частично заключается в их способности обрабатывать запросы, запрашивая записи с ключами, относящимися к определенному диапазону (при этом используется преимущество упорядоченности записей в основном файле в соответствии с последовательностью сортировки). Разреженный индекс обрабатывает подобные запросы также достаточно эффективно — хотя, как правило, все же менее эффективно, чем В-деревья. На интуитивном уровне причина предпочтения В-деревьев (в сравнении с разреженным индексом) заключается в том, что В-дерево можно рассматривать как разреженный индекс для разреженного индекса для разреженного индекса и т.д. (Однако ситуации, когда требуется более трех уровней индексов, встречаются довольно редко.)

В-деревья также зарекомендовали себя относительно неплохо при использовании их в качестве вторичных указателей, когда "ключи" в действительности не определяют ту или иную уникальную запись. Даже если записи с заданным значением для указанных полей вторичного индекса охватывают несколько блоков, мы можем прочитать все эти записи, выполнив столько обращений к блокам, сколько существует блоков, содержащих эти записи, плюс число их предшественников в В-дереве. Для сравнения: если бы эти записи плюс еще одна группа такого же размера оказались хешированными в одном и том же сегменте, тогда поиск любой из этих групп в таблице хеширования потребовал бы такого количества обращений к блокам, которое приблизительно равняется удвоенному числу блоков, требующемуся для размещения любой из этих групп. Возможно, есть и другие причины популярности В-деревьев, например их высокая эффективность в случае, когда к такой структуре одновременно обращается несколько процессов; впрочем, авторы настоящей книги не ставили перед собой задачу освещения подобных вопросов.

Упражнения

11.1. Составьте программу *concatenate* (конкатенация), которая принимает последовательность имен файлов в качестве аргументов и переписывает содержимое этих файлов на стандартное устройство вывода, осуществляя таким образом конкатенацию этих файлов.

11.2. Составьте программу *include* (включить в себя), которая копирует свой вход на свой выход за исключением случая, когда ей встречается строка в форме **#include *имя_файла***; в этом случае программа должна заменить такую строку на содержимое названного файла. Обратите внимание, что включенные файлы также могут содержать операторы #include.

11.3. Как поведет себя программа, о которой было сказано в упражнении 11.2, когда файл "замыкается" на себя?

11.4. Составьте программу *compare* (сравнение), которая **сравнивает** два файла, запись за записью, чтобы выяснить, идентичны ли эти файлы.

***11.5.** Перепишите программу сравнения, о которой говорилось в упражнении 11.4, воспользовавшись **НОП-алгоритмом** из раздела 5.6 для поиска самой длинной общей последовательности записей в обоих файлах.

11.6. Составьте программу *find* (поиск), которая имеет два аргумента, состоящих из строки шаблона и имени файла, и распечатывает все строки файла, содержащие указанную строку шаблона в качестве вложенной строки. Если, например, строкой шаблона является **"ufa"**, а файл представляет собой список слов, тогда *find* распечатывает все слова, содержащие указанные три буквы.

11.7. Составьте программу, которая считывает файл и переписывает записи файла в отсортированной последовательности на свое стандартное устройство **вывода.**

11.8. Какие примитивы предусмотрены в языке Pascal для работы с внешними файлами? Как бы вы усовершенствовали их?

***11.9.** Допустим, используется трехфайловая многофазная сортировка, причем на i-м этапе создается файл с r_i сериями длины l_i. На n-м этапе требуется одна серия из одного файла и ни одной из двух других. Поясните, почему должно соблюдаться каждое из приведенных ниже условий.

а) $l_i = l_{i-1} + l_{i-2}$ **для** $i \geq 1$, где l_0 и l_{-1} — длины серий в двух первоначально занятых файлах;

б) r_i — r_{i-2} - r_{i-1} (или, что эквивалентно, r_{i-2} = Гц + r_i) для i fe 1, где r_0 и r_{-1} — количество серий в двух первоначальных файлах;

в) $r_n = r_{n-1} = 1$ и, следовательно, $r_n, r_{n-1}, ..., r_1$ образуют последовательность чисел Фибоначчи.

*11.10. Какое дополнительное условие нужно добавить к условиям, перечисленным в упражнении **11.9**, чтобы сделать возможным выполнение многофазной сортировки

а) с начальными сериями длиной 1 (т.е. $l_0 = l_{-1}$— **1**);

б) в течение k этапов, но с другими начальными сериями.

Совет. Рассмотрите несколько вариантов, например l_n = 50, l_{n-1} = 31 или l_n - 50, l_{n-1} - 32.

****11.11.** Обобщите упражнения 11.9 и 11.10 на многофазные сортировки с количеством файлов, большим трех.

****11.12.** Покажите, что

а) для сортировки n записей с помощью любого алгоритма внешней сортировки, который использует только одну магнитную ленту в качестве устройства внешней памяти, потребуется времени порядка $\Omega(n^2)$;

б) в случае использования двух магнитных лент в качестве устройств внешней памяти достаточно будет $O(n \log n)$ времени.

11.13. Допустим, имеется внешний файл, содержащий ориентированные дуги $x \to y$, которые образуют ориентированный ациклический граф. Допустим, что не хватает свободного объема оперативной памяти для одновременного хранения всей совокупности вершин или дуг.

а) Составьте программу внешней топологической сортировки, которая распечатывает такую линейно упорядоченную последовательность вершин, что если $x \to y$ представляет собой дугу, то в данной последовательности вершина x появляется до вершины y.

б) Какой будет временная сложность программы как функции от количества обращений к блокам и сколько ей потребуется пространства оперативной памяти?

в) Как поведет себя программа, если ориентированный граф окажется циклическим?

**г) Каково будет минимальное количество обращений к блокам, необходимое для выполнения топологической сортировки при внешнем хранении ориентированного ациклического графа?

11.14. **Допустим,** имеется файл, состоящий из одного миллиона записей, причем каждая запись занимает 100 байт. Блоки имеют длину 1000 байт, а указатель на блок занимает 4 байта. Придумайте вариант организации этого файла с помощью хеширования. Сколько блоков потребуется для хранения таблицы сегментов и самих сегментов?

11.15. Придумайте вариант организации файла, описанного в упражнении 11.14, в виде В-дерева.

11.16. Напишите программы, реализующие операторы RETRIEVE, INSERT, DELETE и MODIFY для

а) **хешированных** файлов;

б) индексированных файлов;

в) файлов в виде В-дерева.

11.17. Составьте программу, выполняющую поиск ft-го наибольшего элемента

а) в файле с разреженным индексом;

б) в файле в виде В-дерева.

***11.18.** Допустим, что на считывание блока, содержащего узел m-**арного** дерева поиска, уходит $a + bm$ миллисекунд. Допустим также, что на обработку каждого узла во внутренней памяти уходит $c + d\log_2 m$ миллисекунд. Если на таком дереве имеется n узлов, то потребуется считать приблизительно $\log_m n$ узлов, чтобы обнаружить нужную запись. Таким образом, общее время, которое потребуется, чтобы найти на дереве нужную запись, составит

$$(\log_m n)(a + bm + c + d\log_2 m) - (\log_2 n)((a + c + bm)/\log_2 m + d)$$

миллисекунд. Сделайте обоснованные оценки значений a, b, c и d и постройте график зависимости этой величины от m. При каком значении m удается достичь минимума?

***11.19.** **В*-дерево** представляет собой В-дерево, в котором каждый внутренний узел заполнен не на половину, а по крайней мере на две третьих. Придумайте схему вставки записи для В*-деревьев, которая позволяет отложить расщепление внутренних узлов до того **момента,** пока не окажутся заполненными два узла-**"брата".** Затем два этих заполненных узла можно разделить на три, каждый из которых будет заполнен на две третьих. Каковы преимущества и недостатки В*-деревьев в сравнении с В-деревьями?

***11.20.** Когда ключ записи представляет собой строку символов, можно сэкономить место в памяти, сохраняя в каждом внутреннем узле В-дерева в качестве разделителя ключей только префикс ключа. Например, слова "cat" и "dog" можно разделить префиксом “d”, или "do", или "dog". Придумайте такой алгоритм вставки для В-дерева, который использует как можно более короткие префиксные разделители ключей.

***11.21.** Допустим, что p-ю часть времени при выполнении операций с определенным файлом занимают вставки и удаления, а оставшуюся $(1 - p)$-ю часть времени занимают операции поиска информации, в которых указывается только одно поле. В записях файла имеются k полей, а i-е поле в операциях поиска указывается с вероятностью q_i. Допустим, что выполнение операции поиска занимает а миллисекунд, если для указанного поля не предусмотрен вторичный индекс, и b миллисекунд, если для этого поля предусмотрен вторичный индекс. Допустим также, что выполнение операций вставки и удаления занимает $c + sd$ миллисекунд, где s — количество вторичных индексов. Определите среднее время выполнение операций как функцию от a, b, c, d, p и q_i и минимизируйте его.

****11.22.** Предположим, что используемый тип ключей допускает возможность их линейного упорядочения (например, ключи являются действительными числами) и известно вероятностное распределение в данном файле ключей с заданными значениями. На основании этих сведений постройте алгоритм поиска ключа в разреженном индексе, превосходящий по эффективности двоичный поиск. Эта статистическая информация используется в одной из схем (которая называется *интерполяционным поиском)* для прогнозирования, в каком именно месте диапазона индексных блоков $B_i, \ldots, B_j$ вероятность появления ключа x является наибольшей. Докажите, что для поиска ключа в среднем было бы достаточно $O(\log \log n)$ обращений к блокам.

11.23. Допустим, что есть внешний файл записей, каждая из которых представляет ребро графа G и стоимость этого ребра.

а) Напишите программу построения остовного дерева минимальной стоимостью для графа G, полагая, что объем основной памяти достаточен для хранения всех вершин графа, но не достаточен для хранения всех его ребер.

б) Какова временная сложность этой программы как функции от количества вершин и ребер?

Совет. Один из подходов к решению этой задачи заключается в сохранении леса связных компонентов в основной памяти. Каждое ребро считывается и обрабатывается следующим образом. Если концы очередного ребра находятся в двух разных компонентах, добавить это ребро и объединить данные компоненты. Если это ребро создает цикл в каком-то из существующих компонентов, добавить это ребро и удалить ребро с наивысшей стоимостью из соответствующего цикла (это ребро может быть текущим). Такой подход подобен алгоритму Крускала, но в нем не требуется сортировки ребер, что весьма существенно для решения данной задачи.

11.24. Допустим, что имеется файл, содержащий последовательность положительных и отрицательных чисел $a_1, a_2, \ldots, a_n$. Напишите программу с временем выполнения $O(n)$, которая находила бы непрерывную вложенную подпоследовательность $a_i, a_{i+1}, \ldots, a_j$, которая имела бы наибольшую сумму $a_i + a_{i+1} + \ldots + a_j$ среди всех вложенных подпоследовательностей такого рода.

Библиографические примечания

Дополнительный материал по внешней сортировке можно найти в книге [65], по структурам внешних данных и их использованию в системах баз данных — там же, а также в [112] и [118]. Многофазная сортировка обсуждается в работе [102]. Схема слияния с шестью буферами, описанная в разделе 11.2, заимствована из [39], а схема с четырьмя буферами — из [65].

Выбор вторичного индекса, упрощенным вариантом которого является упражнение 11.21, обсуждается в работах [72] и [95]. Ссылка на В-деревья впервые встречается в [6]. В [20] приведен обзор различных вариантов В-деревьев, а в [45] обсуждается их практическое применение.

Информацию, касающуюся упражнения 11.12 (сортировка с помощью одной и двух магнитных лент), можно найти в [35]. Тема упражнения 11.22, посвященного интерполяционному поиску, подробно обсуждается в [84] и [124].

Весьма элегантная реализация подхода, предложенного в упражнении 11.23 для решения задачи построения остовного дерева минимальной стоимости, принадлежит В. А. Высоцкому (1960 г., не опубликовано).

ГЛАВА 12

Управление памятью

В этой главе обсуждаются базовые стратегии, позволяющие повторно использовать пространство основной (оперативной) памяти или разделять его между несколькими объектами или процессами, потребность которых в памяти может произвольным образом увеличиваться или уменьшаться. Мы обсудим, например, методы создания связных списков свободного пространства памяти и методы "сборки мусора", при использовании которых доступная память подсчитывается лишь в том случае, когда испытывается нехватка памяти.

12.1. Проблемы управления памятью

В работе компьютерных систем нередко возникают ситуации, когда ограниченными ресурсами основной памяти приходится *управлять,* т.е. разделять между несколькими совместно использующими ее "конкурентами". Программист, который никогда не занимался реализацией системных программ (компиляторов, операционных систем и т.п.), может даже не подозревать о существовании подобных проблем, поскольку действия, связанные с решением этих проблем, зачастую выполняются "за кулисами". Например, программисты, создающие программы на языке Pascal, знают, что процедура ***new*(*p*)** создает указатель *p* на новый объект (динамическую переменную) соответствующего типа. Но откуда берется пространство для хранения этого объекта? Процедура *new* имеет доступ к большой области памяти, называемой *кучей* (**heap**[1] — динамически распределяемая область памяти), которая не используется переменными программы. Из этой области выбирается свободный блок последовательных байтов памяти, достаточный для хранения объекта того типа, на который указывает *p,* а в *p* помещается адрес первого байта этого блока. Но откуда процедуре *new* известно, какие байты памяти "свободны"? Ответ на этот вопрос читатель найдет в разделе 12.4.

Еще более удивительным является то, что происходит, когда значение указателя *p* изменяется то ли после операций присвоения или удаления, то ли после очередного обращения к процедуре ***new*(*p*).** Блок памяти, на который указывал *p,* может оказаться *недоступным* в том смысле, что до него невозможно добраться посредством структур данных вашей программы и, следовательно, нельзя повторно использовать предоставляемое им пространство. С другой стороны, прежде чем изменять *p,* его значение можно было бы скопировать в какую-то другую переменную. В таком случае этот блок памяти по-прежнему оставался бы частью структур данных нашей программы. Но как узнать, что тот или иной блок в области памяти, используемой процедурой ***new*,** больше не требуется вашей программе?

Подход к управлению памятью, используемый в языке Pascal, — лишь один из нескольких возможных. В некоторых случаях (например, в Pascal) одно и то же пространство памяти совместно используют объекты разных размеров. В других случаях все объекты, совместно использующие определенное пространство памяти, имеют

[1] В русскоязычной технической литературе (особенно переводной), посвященной управлению компьютерной памятью, нет единой вполне устоявшейся терминологии, поэтому в данной главе мы будем приводить как термины на русском языке (наиболее точные с нашей точки зрения), так и их англоязычные эквиваленты. — *Прим. ред.*

одинаковые размеры. Это различие, касающееся размера объектов, отражает лишь один из подходов к классификации типов задач управления памятью, с которыми нам приходится сталкиваться. Ниже приведено еще несколько примеров.

1. В языке программирования Lisp пространство памяти делится на *ячейки* (cells), которые, по существу, представляют собой записи, состоящие из двух полей, при этом каждое поле может содержать либо *атом* (объект элементарного типа, например целое число), либо указатель на ячейку. Атомы и указатели имеют одинаковый размер, поэтому для всех ячеек требуется одинаковое количество байт. На основе этих ячеек можно создать любые известные структуры данных. Например, связные списки атомов могут использовать первые поля ячеек для хранения атомов, а вторые поля — для хранения указателей на следующие ячейки в списке. Двоичные деревья также можно представлять с помощью таких ячеек: содержимое первого поля каждой ячейки должно указывать на левого сына, а второго поля — на правого сына. При выполнении Lisp-программы пространство памяти, используемое для хранения ячеек, может оказываться — в разные моменты времени — частью нескольких различных структур; иногда это связано с тем, что ячейка передается из одной структуры в другую, а иногда с тем, что ячейка сразу же освобождается всеми структурами и может, таким образом, использоваться повторно.
2. Файловая система, вообще говоря, делит устройства вторичной памяти (например, диски) на блоки фиксированной длины. Например, UNIX, как правило, использует блоки по 512 байт. Файлы хранятся в совокупности (не обязательно непрерывной) блоков. По мере создания и уничтожения файлов для повторного использования предоставляются блоки вторичной памяти.
3. Типичная мультипрограммная операционная система позволяет нескольким программам одновременно использовать основную память. Каждая программа располагает необходимым ей объемом памяти (известным операционной системе); потребность в памяти является составной частью запроса на обслуживание, выдаваемого в момент запуска программы на выполнение. В то время как в примерах 1 и 2 объекты, совместно использующие память (ячейки и блоки соответственно), имели одинаковые размеры, разным программам требуются разные объемы памяти. Таким образом, когда завершает свою работу программа, использующая, например, 100 Кбайт, вместо нее можно запустить на выполнение две программы, использующие по 50 Кбайт каждая, или одну, которая использует 20 Кбайт, и другую, которая использует 70 Кбайт (в этом случае 10 Кбайт памяти останутся неиспользованными). С другой стороны, 100 Кбайт, освободившиеся в результате завершения работы программы, можно объединить с соседними неиспользуемыми 50 Кбайт, после чего можно запустить на выполнение программу, которой требуется 150 Кбайт памяти. Еще один вариант: ни одна из новых программ не умещается в освободившееся пространство, и 100 Кбайт временно остаются незанятыми.
4. Существует немало языков программирования, например Snobol, APL или SETL, которые выделяют пространство объектам произвольного размера. Этим объектам, которые представляют собой значения, присваиваемые переменным, выделяется блок пространства из более крупного блока памяти, часто называемого *динамической памятью* (heap). Когда значение переменной изменяется, новому значению выделяется пространство в динамической памяти; соответствующим образом корректируется и указатель для этой переменной — теперь он указывает на ее новое значение. Возможно, старое значение переменной уже не требуется, и занимаемое ею пространство может использоваться повторно. Однако в таких языках, как Snobol или SETL, присвоения вида $A = B$ реализуются путем изменения указателя: указатель для A теперь указывает на тот же объект, на который указывает указатель для B. Если для A или B выполнено новое присвоение, предыдущий объект не освобождается, а его пространство не подлежит повторному использованию.

Пример 12.1. На рис. 12.1, а показана динамическая память, которая может использоваться Snobol-программой с тремя переменными: *A, B* и *C*. Значение любой переменной в Snobol является строкой символов, и в данном случае значением *A и B* является 'OH HAPPY DAY', а значением *C* — 'PASS THE SALT'[1].

Мы хотим хранить символьные строки с помощью указателей на блоки в динамической памяти. Первые 2 байта каждого из этих блоков (число 2 является типичным значением, которое можно изменить) выделены для целого числа, указывающего длину строки. Например, длина строки 'OH HAPPY DAY' равна 12 байтам (с учетом пробелов между словами), поэтому значение *A* (и *B)* занимает 14 байт.

Если значение *B* заменить на 'OH HAPPY DAZE'[2], то необходимо подыскать в динамической памяти пустой блок длиной 15 байтов, включая 2 байта для указания длины. Далее выполняется корректировка указателя для переменной *B* так, чтобы он указывал на новое значение (рис. 12.1, б). Блок, содержащий целое число 12 и строку 'OH HAPPY DAY', еще понадобится, поскольку на него указывает *A*. Если же изменить и значение *A*, этот блок будет уже бесполезен и подлежит повторному использованию. Мы еще вернемся в этой главе к вопросу о том, на основании каких признаков можно с уверенностью говорить об отсутствии указателей на такой блок. □

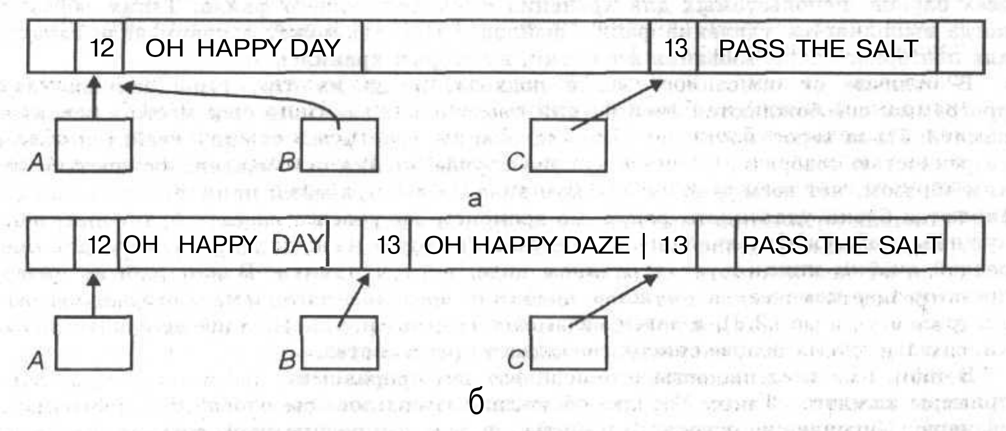


Рис. 12.1. Строковые переменные в динамической памяти

В четырех приведенных выше примерах различия заметны, по крайней мере, по двум взаимно перпендикулярным "измерениям". Первый вопрос: имеют ли одинаковую длину объекты, совместно использующие память. В первых двух примерах (Lisp-программы и хранение файлов) объекты, которыми в одном случае являются Lisp-ячейки, а в другом — блоки, содержащие части файлов, имеют одинаковые размеры. Этот факт позволяет идти на определенные упрощения при решении задачи управления памятью. Например, в реализации Lisp область памяти делится на участки, каждый из которых может содержать ровно одну ячейку. Задача управления заключается в том, чтобы найти пустые участки, в которых могли бы разместиться вновь созданные ячейки. Аналогично, во втором примере диск делится на блоки одинакового размера; каждый блок предназначен для хранения определенной части одного файла: никогда не используется один блок для хранения частей нескольких файлов — даже если файл заканчивается посредине блока.

В третьем и четвертом примерах, где описывается выделение памяти в мультипрограммной системе и управление динамической памятью в языках, имеющих дело с переменными, значения которых представляют собой "большие" объекты, напро-

[1] Значения переменных переводятся как "О, счастливый день" и "Передайте соль". — *Прим. перев.*

[2] В данном случае значение переменной *B* обозначает "О, счастливое изумление". — *Прим. перев.*

тив, говорится о выделении пространства блоками разных размеров. Это требование создает определенные проблемы, которые отсутствуют при выделении блоков фиксированной длины. Например, мы стремимся избежать *фрагментации* — ситуации, когда неиспользуемое пространство достаточно велико, но настолько раздроблено, что найти свободный участок для размещения крупного объекта не представляется возможным. Подробнее об управлении динамической памятью мы поговорим в разделах 12.4 и 12.5.

Второй важный вопрос заключается в том, каким именно способом, явным или неявным, выполняется *сборка мусора* (garbage collection — чистка памяти) — "симпатичный" термин, обозначающий возврат в систему неиспользуемого пространства. Иными словами, речь идет о том, должна ли программа выдавать для этого специальную команду, или "сборка мусора" выполняется в ответ на запрос выделения пространства, который в противном случае оказалось бы невозможным выполнить. Рассмотрим управление файлами. Когда выполняется удаление файла, файловой системе известны блоки, в которых хранился этот файл. Например, в файловой системе можно было бы хранить адреса одного или нескольких "главных блоков" каждого файла, созданного в этой системе. В этих блоках хранится список адресов всех блоков, используемых для хранения соответствующего файла. Таким образом, когда выполняется удаление файла, файловая система может в явном виде вернуть для повторного использования все блоки, в которых хранился этот файл.

В отличие от описанного выше подхода, когда из структуры данных Lisp-программы освобождаются ячейки, они продолжают занимать свое место в основной памяти. Из-за вероятности наличия нескольких указателей на одну ячейку нельзя с уверенностью говорить, в каком именно случае ячейка освобождена полностью; таким образом, нет возможности в явном виде собирать ячейки примерно так, как собираются блоки удаленного файла. Со временем все участки памяти будут выделены нужным и ненужным ячейкам, и следующий запрос на выделение памяти для очередной ячейки инициирует (в неявном виде) чистку памяти. В этот момент интерпретатор Lisp помечает все нужные ячейки (с помощью алгоритма, который мы рассмотрим в разделе 12.3), а затем связывает все блоки, содержащие ненужные ячейки, создавая на их основе список свободного пространства.

В табл. 12.1 сведены четыре описанных вида управления памятью и приведены примеры каждого из них. Мы уже обсуждали использование блоков фиксированного размера. Управление основной памятью в мультипрограммной системе является примером явной "утилизации" блоков переменной длины: когда программа завершает работу, операционная система, зная, какая область памяти была выделена этой программе, и зная, что никакая другая программа не могла использовать это пространство, тотчас же делает это пространство доступным другой программе.

Таблица 12.1. Примеры четырех стратегий управления памятью

		Утилизация неиспользуемого пространства	
		явная	"сборка мусора"
Размер блока	фиксированный	Файловая система	Lisp
	переменный	Мультипрограммная система	Snobol

Управление динамической памятью в языке Snobol и многих других языках программирования является примером использования блоков переменной длины при чистке памяти. Как и в случае языка Lisp, типичный интерпретатор Snobol не пытается "отдавать" блоки памяти до тех пор, пока не столкнется с нехваткой памяти. В этот момент интерпретатор Snobol (как и интерпретатор Lisp) приступает к "сборке мусора". Правда, у интерпретатора Snobol появляется дополнительная возможность:

строки "тасуются" в области динамической памяти с целью уменьшения фрагментации; кроме того, свободные смежные блоки объединяются, образуя более крупные блоки. Обратите внимание, что две последние возможности в Lisp не используются.

12.2. Управление блоками одинакового размера

Представим, что у нас есть программа, манипулирующая ячейками, каждая из которых состоит из пары полей; каждое поле может быть указателем на какую-либо ячейку или содержать "атом". Эта ситуация, конечно, характерна для любой программы, написанной на Lisp, однако такую программу можно написать практически на любом языке программирования, в том числе и на языке Pascal, если мы определим ячейки как записи (тип record). Пустые ячейки, которые можно включить в какую-либо структуру данных, указаны в списке свободного пространства, а каждая переменная программы представляется указателем на соответствующую ячейку. Указываемая ячейка может быть одной из ячеек более крупной структуры данных.

Пример 12.2. На рис. 12.2 приведен один из возможных вариантов структуры сети ячеек. Здесь *A, B и C* — переменные, строчными буквами обозначены атомы. Обратим внимание на некоторые интересные явления. На ячейку, содержащую атом *a*, указывает переменная *A* и еще одна ячейка. На ячейку, содержащую атом *c*, указывают две разные ячейки. Ячейки, содержащие атомы *g* и А, необычны в том отношении, что хотя они и указывают друг на друга, до них невозможно "добраться" из переменных *A, B* и С; более того, они отсутствуют в списке свободного пространства. □

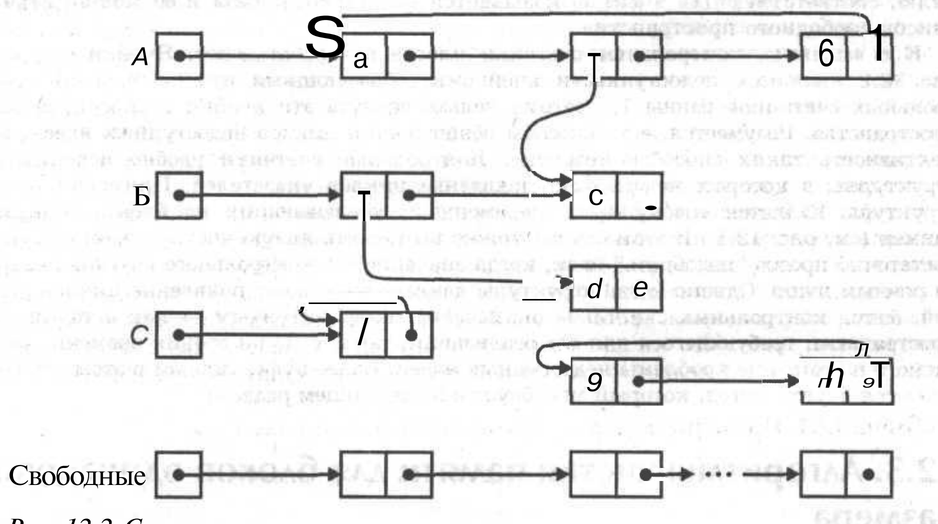


Рис. 12.2. Сеть ячеек

Допустим, что при выполнении программы из списка свободного пространства могут периодически изыматься новые ячейки. Например, может понадобиться заменить нуль-указатель в ячейке с атомом с (см. рис. 12.2) указателем на новую ячейку, которая содержит атом *i* и нуль-указатель. Эта ячейка будет изъята из вершины списка свободного пространства. Возможно также, что время от времени указатели будут изменяться таким образом, что переменные программы станут освобождать занятые ими ячейки, как это случилось, например, с ячейками, содержащими *g* и А, на рис. 12.3. Например, ячейка, содержащая *c*, может в какой-то момент указывать на ячейку, содержащую *g*. Или еще один пример: значение переменной *B* может в какой-то момент измениться, что приведет (если не изменится что-либо еще) к освобождению ячейки, на которую сейчас на рис. 12.2 указывает *B*, а также ячейки, со-

держащей *d* и *e* (но не ячейки, содержащей *c,* поскольку она по-прежнему будет доступна из *A).* Ячейки, на которые невозможно перейти ни из одной переменной и которые отсутствуют в списке свободного пространства, называются *недоступными.*

Когда ячейки освобождаются и, следовательно, уже не требуются программе, было бы неплохо, если бы они каким-то образом возвращались в список свободного пространства и могли после этого использоваться повторно. Если такие ячейки не будут "утилизироваться", рано или поздно возникнет ситуация, когда программа не использует ни одной ячейки и, тем не менее, ей требуется очередная ячейка из свободного пространства памяти, в то время как список свободного дространства пуст. Именно в этот момент системе потребуется время на выполнение "сборки мусора". Этап чистки памяти выполняется "неявным" образом в том смысле, что мы не выдавали никакого специального запроса на получение дополнительных ячеек памяти.

Контрольные счетчики

Весьма привлекательным подходом к выявлению недоступных ячеек является включение в каждую ячейку так называемого *контрольного счетчика,* или *счетчика ссылок* (reference count), т.е. целочисленного поля, значение которого равняется количеству указателей на соответствующую ячейку. Работу такого счетчика обеспечить несложно. С момента создания указателя на какую-либо ячейку значение контрольного счетчика для этой ячейки увеличивается на единицу. Когда переназначается ненулевой указатель, значение контрольного счетчика для указываемой ячейки уменьшается на единицу. Если значение контрольного счетчика становится равным нулю, соответствующая ячейка оказывается невостребованной и ее можно вернуть в список свободного пространства.

К сожалению, контрольные счетчики иногда не срабатывают. Ячейки с *g* и *h* на рис. 12.2 являются недоступными ячейками, образующими цикл. Значения их контрольных счетчиков равны 1, поэтому нельзя вернуть эти ячейки в список свободного пространства. Разумеется, есть способы обнаружения циклов недоступных ячеек, но эффективность таких способов невелика. Контрольные счетчики удобно использовать в структурах, в которых невозможно появление циклов указателей. Примером подобной структуры является совокупность переменных, указывающих на блоки, содержащие данные (см. рис. 12.1). В этом случае можно выполнять явную чистку памяти. Для этого достаточно просто "подобрать" блок, когда значение его контрольного счетчика становится равным нулю. Однако когда структуры данных допускают появление циклов указателей, метод контрольных счетчиков оказывается неэффективным — как с точки зрения пространства, требующегося для его реализации, так и с точки зрения времени, затрачиваемого на решение проблемы недоступных ячеек. Более эффективным в этом случае оказывается другой метод, который мы обсудим в следующем разделе.

12.3. Алгоритмы чистки памяти для блоков одинакового размера

Рассмотрим алгоритм, позволяющий определить, к каким ячейкам, показанным на рис. 12.2, возможен доступ из переменных программы. Точную постановку задачи можно выполнить, определив тип ячейки, который в языке Pascal будет представлен определенным типом записи. Четыре варианта, которые мы обозначим *PP, PA, AP* и *AA,* определяются тем, какие из двух полей данных являются указателями, а какие — атомами *(P* обозначает поле указателя (от pointer — указатель), *A* — поле атома). Например, *PA* означает, что левое поле является указателем, а правое поле — атомом. Дополнительное булево поле в ячейках, называемое *mark* (метка), показывает, является ли данная ячейка свободной, т.е. установив при "сборке мусора" для поля *mark* значение true, мы помечаем соответствующую ячейку как доступную для последующего использования. Определения типов данных показаны листинге 12.1.

Листинг 12.1. Объявления типов ячеек

```
type
    atomtype = {один из подходящих типов данных для атомов,
                желательно того же размера, что и указатели }
    patterns = (PP,PA,AP,AA);
    celltype = record
        mark: boolean;
        case pattern: patterns of
            PP: (left: ↑celltype; right: ↑celltype);
            PA: (left: ↑celltype; right: atomtype);
            AP: (left: atomtype; right: ↑celltype);
            AA: (left: atomtype; right: atomtype);
    end;
```

Предполагаем, что имеется массив ячеек, занимающий большую часть памяти, и определенная совокупность переменных, которые представляют собой указатели на ячейки. С целью упрощения также предполагаем, что есть только одна переменная, называемая *source* (исходная), которая указывает на ячейку (хотя обобщение на случай многих переменных не представляет особой трудности).[1] Таким образом, объявляем

```
    var
        source: ↑celltype;
        memory: array [1..memorysize] of celltype;
```

Чтобы пометить ячейки, доступные переменной *source,* сначала отменяем пометку всех ячеек (независимо от их доступности или недоступности), просматривая массив *memory* (память) и устанавливая для поля *mark* значение false. Затем выполняем поиск в глубину по дереву ячеек, исходя из ячейки *source* и помечая все посещаемые ячейки. Посещенные ячейки — именно те, которые доступны из программы и, следовательно, используются программой. Затем просматриваем массив *memory* и добавляем в список свободного пространства все непомеченные ячейки. В листинге 12.2 представлена процедура *dfs,* выполняющая поиск в глубину; вызов *dfs* осуществляется процедурой *collect* (выбор), которая отменяет пометку всех ячеек, а затем маркирует доступные ячейки, вызывая *dfs.* Мы не показываем программу, создающую связанный список свободного пространства, из-за особенностей языка Pascal. Например, несмотря на то что можно было бы связать свободные ячейки с помощью либо всех левых, либо всех правых полей ячеек, поскольку предполагается, что указатели и атомы имеют одинаковый размер, но мы не имеем права заменять атомы на указатели в ячейках типа *AA.*

Листинг 12.2. Алгоритм маркировки доступных ячеек

```
(1) procedure dfs ( currentcell: ↑celltype );
        { Если текущая ячейка помечена, не делать ничего.
            В противном случае она помечается и вызывается dfs
            для всех ячеек, на которые указывает текущая ячейка }
        begin
(2)         with currentcell↑ do
(3)             if mark = false then begin
(4)                 mark := true;
```

[1] **В каждом языке программирования должен быть предусмотрен метод представления текущей совокупности переменных. Для этого подходит любой из методов, обсуждавшихся в главах 4 и 5. Например, в большинстве реализаций для хранения таких переменных используется хеш-таблица.**

```
(5)                 if (pattern = PP) or (pattern = PA) then
(6)                     if left <> nil then
(7)                         dfs(left);
(8)                 if (pattern - PP) or (pattern = AP) then
(9)                     if right <> nil then
(10)                        dfs(right)
            end
        end; { dfs }

(11)procedurecollect;
        var
            i: integer;
        begin
(12)        for i  := 1 to memorysize do
                    { отмена маркировки всех ячеек }
(13)            memory[i] .mark:= false;
(14)        dfs(source); { пометить доступные ячейки }
(15)        {в этом месте должен быть код для сбора доступных ячеек}
        end; { collect }
```

Сборка на месте

Алгоритм, представленный в листинге 12.2, страдает небольшим изъяном. В вычислительной среде с ограниченным объемом памяти у нас может не оказаться достаточно места для хранения стека, требующегося для рекурсивных вызовов *dfs*. Как уже указывалось в разделе 2.6, каждый раз, когда *dfs* вызывает сама себя, Pascal (или любой другой язык, допускающий рекурсию) создает для данного вызова *dfs* так называемую ***активационную запись***. Вообще говоря, в такой активационной записи предусмотрено место для параметров и переменных, локальных по отношению к процедуре, причем для каждого вызова требуется свой собственный экземпляр активационной записи. Кроме того, в каждой активационной записи нужно указывать *адрес возврата* — место программы, в которое необходимо передать управление после завершения рекурсивного вызова процедуры.

При вызове процедуры *dfs* требуется лишь место для параметра и адреса возврата. Однако если бы вся память была связана в цепочку, исходящую из ячейки *source* (и, следовательно, количество активных вызовов *dfs* в какой-то момент времени стало равным длине этой цепочки), для стека активационных записей могло бы потребоваться значительно больше пространства, чем имеется в памяти. И если бы свободного пространства не оказалось, было бы невозможно выполнять маркировку освободившихся ячеек.

К счастью, разработан замечательный алгоритм, известный под названием *алгоритма Дойча — Шорра — Уэйта* (Deutsch — Schorr — Waite), который выполняет маркировку на месте. Читатель должен проверить сам, что последовательность ячеек, на которых был выполнен (но еще не завершен) вызов *dfs,* действительно образует путь от *source* до ячейки, на которой был сделан текущий вызов *dfs.* Поэтому можно воспользоваться нерекурсивной версией процедуры *dfs,* а вместо стека активационных записей, предназначенного для отслеживания пути ячеек от *source* до ячейки, анализируемой в данный момент, можно использовать поля указателей вдоль этого пути, которые и будут формировать этот путь. Таким образом, каждая ячейка на этом пути, за исключением последней, содержит или в поле *left* (влево), или в поле *right* (вправо) указатель на ее предшественника — ячейку, расположенную ближе к ячейке *source*. Мы опишем алгоритм Дойча — Шорра — Уэйта, использующий еще одно однобитовое поле, которое называется *back* (назад). Поле *back* имеет перечислимый тип (*L, R)* и говорит о том, какое из полей, *left* или *right,* указывает на предшественника. Далее покажем, как информацию, хранящуюся в поле *back,*

можно включить в поле *pattern* (шаблон), чтобы избежать использования дополнительного места в ячейках.

Эта новая процедура, выполняющая нерекурсивный поиск *в* глубину (назовем ее *nrdfs*[1]), использует указатель *current* (текущий) для указания на текущую ячейку, а указатель *previous* (предыдущий) — для указания на предшественника текущей ячейки. Переменная *source* указывает на ячейку *source*1, которая содержит указатель только в своем правом поле.[2] До выполнения маркировки в ячейке *sourcel* значение поля *back* устанавливается равным *R, а* ее правое поле указывает на самого себя. На ячейку, на которую обычно указывает *sourcel,* теперь указывает ячейка *current,* а на *sourcel* указывает ячейка *previous.* Операция маркировки прекращается в том случае, когда указатели *current* – *previous,* это может произойти лишь при условии, если обе ячейки *current* и *previous* указывают на *sourcel,* т.е. когда уже просмотрена вся структура.

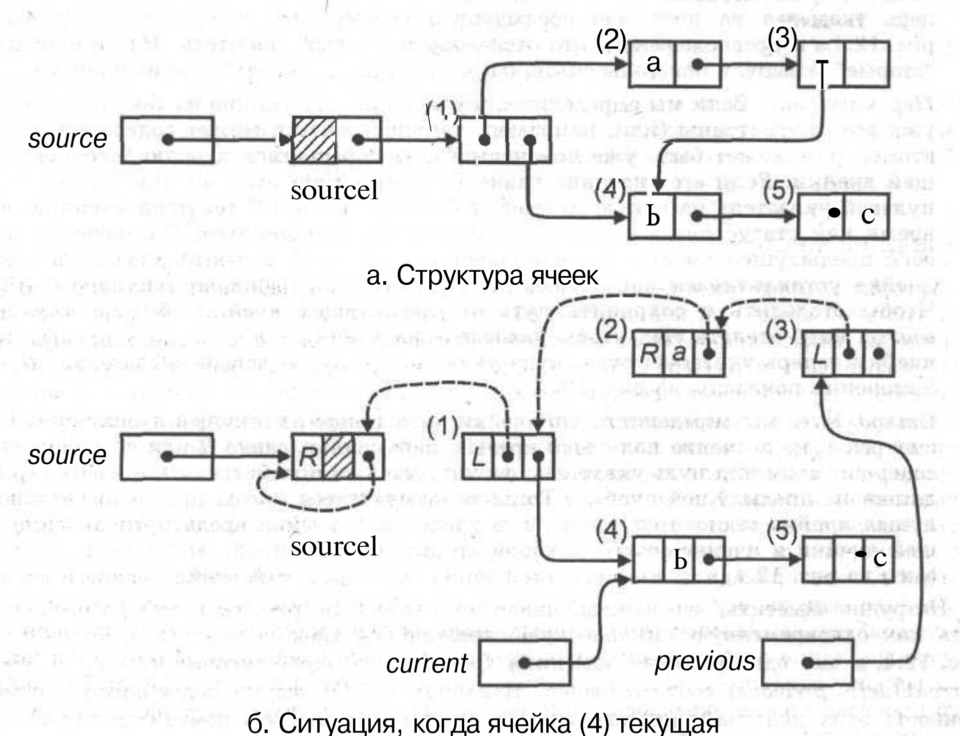


Рис. 12.3. Использование указателей для представления обратного пути к ячейке source

Пример 12.3. На рис. 12.3, а показан возможный вариант структуры ячеек, исходящих из ячейки *source.* Если выполнить поиск в глубину этой структуры, мы посетим ячейки (1), (2), (3) и (4) — именно в такой последовательности. На рис. 12.3, б показано изменение указателей, выполненное в момент, когда ячейка (4) является текущей. Здесь показаны значения поля *back,* а поля *mark* и *pattern* не отображены. Текущий путь от ячейки (4) к ячейке (3) осуществляется через указатель ячейки *previous,* дальнейший путь к ячейкам (2) и (1) и обратно к ячейке *sourcel* показан пунктирными указателями. Например, у ячейки (1) значение поля *back* равно *L,* поскольку на рис. 12.3,а поле *left* этой ячейки содержало указатель на ячейку (2), а

---

[1] Это название — сокращение от *nonrecursive depth-firstseach* (нерекурсивный поиск в глубину). — *Прим. перев.*

[2] Такое несколько неуклюжее решение связано с особенностями языка Pascal.

теперь используется для хранения части пути. Теперь поле *left* ячейки (1) указывает назад, а не вперед по пути; однако мы восстановим этот указатель, когда поиск в глубину перейдет с ячейки (2) обратно на ячейку (1). Аналогично, в ячейке (2) значение поля *back* равно *R,* а правое поле указывает в обратном направлении на ячейку (1), а не вперед на ячейку (3), как это показано на рис. 12.3, а. □

При осуществлении поиска в глубину выполняются три основных шага.

1. *Продвижение вперед.* Если мы определили, что текущая ячейка имеет один или несколько ненулевых указателей, нужно перейти на первый из них, т.е. следовать указателю в поле *left* или, если такового не имеется, — указателю в поле *right.* Теперь надо преобразовать ячейку, на которую указывает текущая ячейка, в "новую" текущую, а "старую" текущую ячейку — в предыдущую. Чтобы облегчить поиск обратного пути, нужно сделать так, чтобы указатель, которому мы только что следовали, теперь указывал на прежнюю предыдущую ячейку. Эти изменения показаны на рис. 12.4,а в предположении, что отслеживался левый указатель. На этом рисунке "старые" указатели показаны сплошными линиями, а "новые" — пунктирными.
2. *Переключение.* Если мы определили, что ячейки, исходящие из текущей ячейки, уже все просмотрены (или, например, текущая ячейка может содержать только атомы, или может быть уже помеченной), то обращаемся к полю *back* предыдущей ячейки. Если его значение равно *L*, а поле *right* этой ячейки содержит ненулевой указатель на какую-то ячейку *C,* тогда делаем *C* текущей ячейкой, в то время как статус предыдущей ячейки остается неизменным. Но значение поля *back* предыдущей ячейки устанавливается равным *R,* а левый указатель в этой ячейке устанавливаем так, чтобы он указывал на прежнюю текущую ячейку. Чтобы отследить и сохранить путь от предыдущей ячейки обратно к ячейке *source,* надо сделать так, чтобы указатель на ячейку С в поле *right* предыдущей ячейки теперь указывал туда, куда указывал ранее ее левый указатель. Все эти изменения показаны на рис. 12.4, б.
3. *Отход.* Если мы определили, что ячейки, исходящие из текущей ячейки, уже просмотрены, но значение поля *back* предыдущей ячейки равно *R* или *L,* а поле *right* содержит атом или нуль-указатель, значит, мы уже просмотрели все ячейки, исходящие из предыдущей ячейки. Тогда осуществляется отход, при котором предыдущая ячейка становится текущей, а следующая ячейка вдоль пути от предыдущей ячейки к ячейке *source* — новой предыдущей ячейкой. Эти изменения показаны на рис. 12.4, в (здесь значение поля *back* предыдущей ячейки равно *R).*

Нетрудно заметить, что каждый шаг, показанный на рис. 12,4, можно рассматривать как одновременную циклическую перестановку трех указателей. Например, на рис. 12.4, а мы одновременно заменили *(previous, current, current↑.left)* на *(current, current↑.left, previous)* соответственно. В данном случае важно подчеркнуть одновременность этих действий: местоположение ячейки *current↑.left* изменится только при присваивании нового значения *current.* Чтобы выполнить эти модификации указателей, используется процедура *rotate* (циклический сдвиг), показанная в листинге 12.3.

Листинг 12.3. **Процедура модификации указателей**

```
procedure rotate ( var p1, p2, p3: ↑celltype );
    var
        temp: ↑celltype;
    begin
        temp:= p1;
        p1:= p2;
        p2:= p3;
        p3:= temp
    end; { rotate }
```

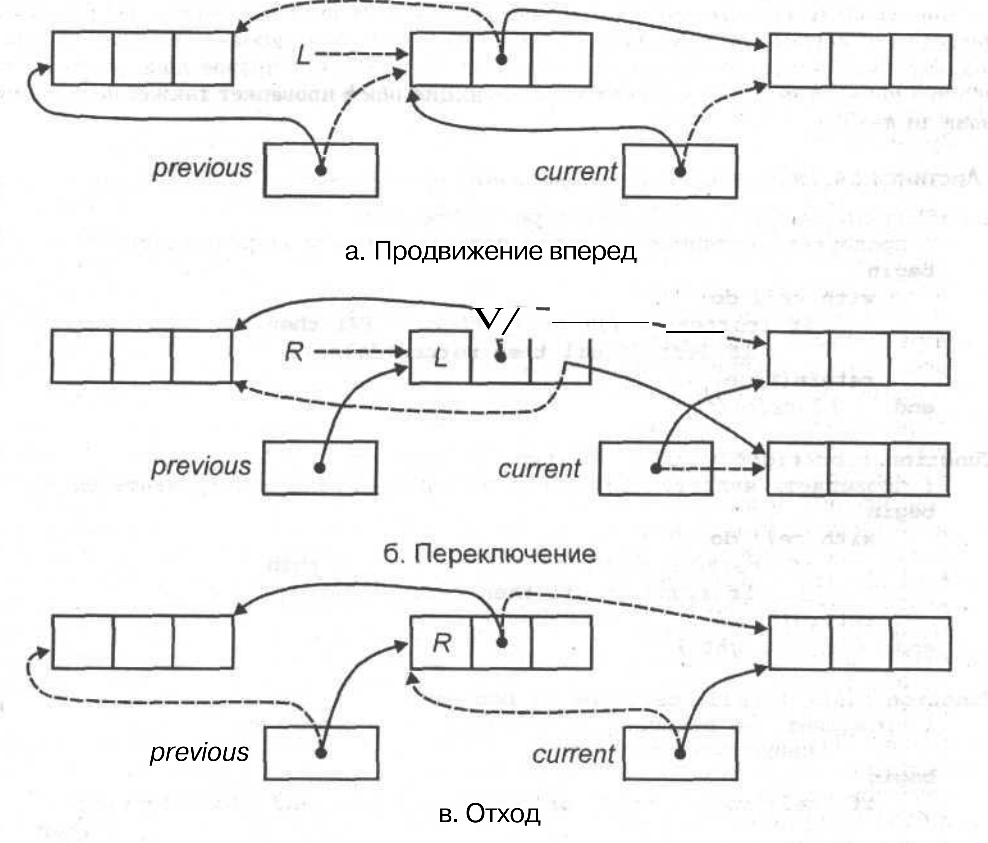


Рис. 12.4. Три основных шага поиска в глубину

Теперь вернемся к написанию нерекурсивной процедуры маркировки *nrdfs.* Эта процедура представляет собой один из тех запутанных процессов, которые проще всего понять, если при их написании использовать метки и операторы безусловного перехода goto. Для процедуры *nrdfs* возможны два "состояния": продвижение вперед, представленное меткой state1, и отход, представленный меткой state2. Поначалу, а также в тех случаях, когда мы перешли на новую ячейку (либо в результате шага продвижения вперед, либо в результате шага переключения), мы переходим в первое состояние. В этом состоянии мы пытаемся выполнить еще один шаг продвижения вперед и "отходим" или "переключаемся" лишь в том случае, если оказываемся заблокированными. Можно оказаться заблокированными по двум причинам: (1) ячейка, к которой только что обратились, уже помечена; (2) в данной ячейке отсутствуют ненулевые указатели. Оказавшись заблокированными, переходим во второе состояние — состояние отхода.

Переход во второе состояние возможен в тех случаях, когда мы отходим или когда не можем оставаться в состоянии продвижения вперед, так как оказались заблокированными. В состоянии отхода проверяем, отошли ли обратно на ячейку *sourcel.* Как уже было сказано выше, мы распознаем эту ситуацию по выполнению условия *previous = current;* в этом случае переходим в состояние 3 (метка state3), т.е. практически закончили маркировку ячеек. В противном случае принимается решение либо отступить и остаться в состоянии отхода, либо переключиться и перейти в состояние продвижения вперед.

Код процедуры *nrdfs* показан в листинге 12.4. В этой программе используются функции *blockleft* (блокировка слева), *blockright* (блокировка справа) и *block* (блокировка), которые проверяют, содержит ли левое или правое поле ячейки (или оба эти поля) атом или нуль-указатель. Функция *block* проверяет также, не маркирована ли ячейка.

Листинг 12.4. Нерекурсивный алгоритм маркировки ячеек

```
function blockleft ( cell: celltype ): boolean;
    { проверяет, является ли левое поле атомом или нуль-указателем }
    begin
        with cell do
            if (pattern = PP) or (pattern = PA) then
                if left <> nil then return(false);
        return(true)
    end; { blockleft }

function blockright ( cell: celltype ): boolean;
    { проверяет, является ли правое поле атомом или нуль-указателем }
    begin
        with cell do
            if (pattern = PP) or (pattern = AP) then
                if right <> nil then return(false);
        return(true)
    end; { blockright }

function block ( cell: celltype ): boolean;
    { проверяет, не помечена ли ячейка и не содержит ли
            ненулевые указатели }
    begin
        if (cell.mark = true) or blockleft(cell) and blockright(cell)
                                                                  then
            return(true)
        else
            return(false)
    end; { block }

procedure nrdfs; { помечает ячейки, доступные из ячейки source }
    var
        current,previous: ↑celltype;
    begin { инициализация }
        current:= source1↑.right;
                { ячейка, на которую указывает source1 }
        previous:= source1; { теперь previous указывает на source1 }
        source1↑.back:= R;
        source1↑.right:= source1; { source1 указывает сама на себя }
        source1↑.mark:= true;
    state1: { продвижение вперед }
        if block(current↑) then begin { подготовка к отходу }
            current↑.mark:= true;
            goto state2
        end
        else begin { пометить и продвинуться вперед }
            current↑.mark:= true;
            if blockleft(current↑) then begin
```

```
                    { следование правому указателю }
                  current↑.back:= R;
                  rotate(previous, current, current↑.right);
                    { реализация изменений как на рис. 12.4,а,
                        но следуя правому указателю }
                  goto state1
              end
              else begin { следование левому указателю }
                  current↑.back:= L;
                  rotate(previous, current, current↑.left);
                    { реализация изменений как на рис. 12.4,а }
                  goto state1
              end
          end;
  state2: { завершение, отход или переключение }
      if previous = current then { завершение }
          goto state3
      else if (previous↑.back:= L) and
              not blockright(previous↑) then begin { переключение }
                  previous↑.back:= R;
                  rotate(previous↑.left, current, previous↑.right);
                      { реализация изменений как на рис. 12.4,б }
                  goto state1
              end
      else if previous↑.back = R then { отход }
          rotate(previous,previous↑.right, current)
                  { реализация изменений как на рис. 12.4,в }
      else { previous↑.back = L }
          rotate(previous,previous↑.left, current);
              { реализация изменений как на рис. 12.4,в,
                  но поле left предыдущей ячейки включено в путь }
      goto state2
      end;
  state 3: { здесь необходимо вставить код для связывания
                  непомеченных ячеек в список свободной памяти }
  end; { nrdfs }
```

Алгоритм Дойча — Шорра — Уэйта без использования поля back

Возможно, хотя и маловероятно, что дополнительный бит, используемый в ячейках для поля *back,* может вызвать потребность в дополнительном байте или даже дополнительном машинном слове для хранения содержимого этого поля. Но оказывается, что без этого дополнительного бита вполне можно обойтись, по крайней мере при программировании на языке, который, в отличие от языка Pascal, позволяет использовать биты поля *pattern* для целей, отличных от тех, для выполнения которых они изначально предназначались. Весь "фокус" заключается в том, что если вообще используется поле *back,* то, поскольку рассматриваемая ячейка находится на обратном пути к ячейке *source1*, поле *pattern* может иметь лишь вполне определенные значения. Если, например, *back = L,* тогда известно, что поле *pattern* должно иметь значение *PP* или *PA,* поскольку очевидно, что поле *left* содержит указатель на какую-либо ячейку. К аналогичному выводу можно прийти и тогда, когда *back = R.* Таким образом, если взять два бита для представления как значения поля *pattern,* так и (в случае необходимости) значения поля *back,* можно закодировать требуемую информацию так, как показано, например, в табл. 12.2.

Таблица 12.2. Интерпретация двух битов как значений полей *pattern* и *back*

Код	На пути к ячейке source1	Не на пути к *source*1
00	*back = L, pattern = PP*	*pattern = PP*
01	*back = L, pattern = PA*	*pattern = PA*
10	*back = R, pattern = PP*	*pattern* = AP
11	*back = R, pattern = AP*	*pattern* = AA

Читатель наверное заметил, что в программе, представленной в листинге 12.3, нам всегда известно, используется ли поле *back,* и, таким образом, всегда можно сказать, какая из интерпретаций, показанных в табл. 12.2, подходит в данном случае. Попросту говоря, когда ячейка *current* указывает на ту или иную запись, то поле *back* в этой записи не используется; когда же ячейка *previous* указывает на запись — значит, используется. Разумеется, при перемещении этих указателей необходимо откорректировать соответствующие коды. Если, например, ячейка *current* указывает на ячейку с битами 10, что в соответствии с табл. 12.2 интерпретируется как *pattern = AP,* и выполняется продвижение вперед (в этом случае *previous* будет уже указывать на эту ячейку), надо положить *back = R,* как только в правом поле окажется указатель, а соответствующие биты сделать равными 11. Обратите внимание: если у ячейки *pattern — AA* (этот вариант не представлен в среднем столбце табл. 12.2), тогда нет необходимости, чтобы *previous* указывала на эту ячейку, поскольку для продвижения вперед нет указателей.

12.4. Выделение памяти для объектов разного размера

Рассмотрим теперь управление динамической памятью (кучей), когда существуют указатели на выделенные блоки (как показано на рис. 12.1). Эти блоки содержат данные того или иного типа. На рис. 12.1, например, эти данные являются строками символов. Несмотря на то что тип данных, хранящихся в динамической памяти, вовсе не обязательно должен быть символьной строкой, мы полагаем, что эти данные не содержат указателей на те или иные адреса в динамической памяти.

У задачи управления динамической памятью есть свои аспекты, которые делают ее, с одной стороны, легче, а с другой — тяжелее задачи управления структурами списков ячеек одинаковой длины, которой был посвящен предыдущий раздел. Основной фактор, который облегчает задачу управления динамической памятью, заключается в том, что маркировка использованных блоков не является рекурсивным процессом: нужно лишь следовать внешним указателям на динамическую память и помечать указываемые блоки. Нет необходимости выполнять просмотр связной структуры или что-то наподобие алгоритма Дойча — Шорра — Уэйта.

С другой стороны, здесь управление списком свободного пространства не так просто, как описано в разделе 12.3. Представим, что свободные участки памяти (например, на рис. 12.1, а показаны три таких свободных участка) связаны друг с другом так, как предлагается на рис. 12.5. Здесь мы видим динамическую память из 3 000 байт, поделенных на пять блоков. Два блока из 200 и 600 байт содержат значения *X* и *Y*. Остальные три блока свободны и связаны в цепочку, исходящую из ячейки *avail* (доступно) заголовка свободного пространства.

Если мы хотим, чтобы эти свободные блоки можно было найти, когда программе потребуется память для хранения новых данных, необходимо сделать следующие предположения, касающиеся блоков динамической памяти (эти предположения распространяются на весь данный раздел).

1. Каждый блок имеет объем, достаточный для хранения
   а) *счетчика,* указывающего размер блока (в байтах или машинных словах в соответствии с конкретной компьютерной системой);

б) указателя (для связи данного блока со свободным пространством);

в) бита заполнения, указывающего, является ли данный блок пустым; этот бит называется битом *full/empty* (заполнено/свободно) или *used/unused* (используется/не используется).

2. В свободном (пустом) блоке слева (со стороны меньших значений адреса) содержатся счетчик, указывающий длину блока, бит заполнения, содержащий значение 0 (свидетельствует о том, что данный блок — пустой), указатель на следующий свободный блок.

3. Блок, в котором хранятся данные, содержит (слева) счетчик, бит заполнения, содержащий значение 1 (свидетельствует о том, что данный блок занят), и собственно данные.[1]

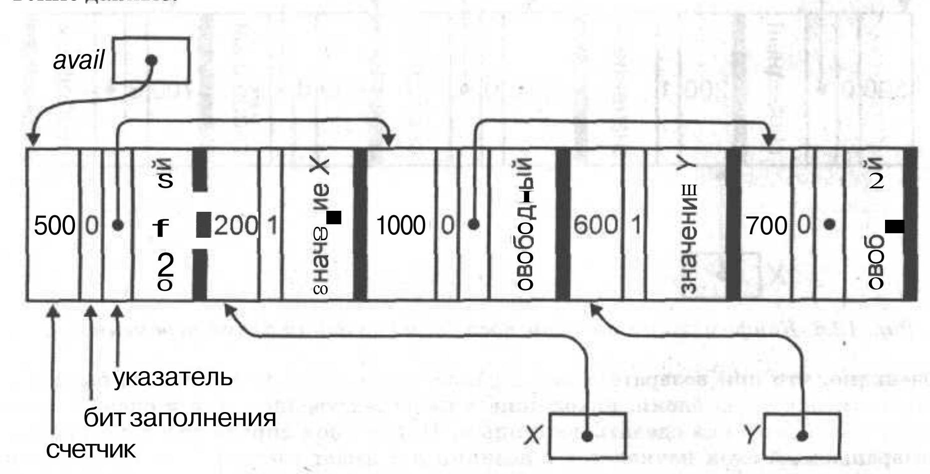


Рис. 12.5. Динамическая память со списком свободного пространства

Из перечисленных предположений следует один интересный вывод: блоки должны "уметь" хранить в одном и том же месте иногда данные (в тех случаях, когда блок используется), а во всех остальных случаях указатели (когда блок не используется). В результате оказывается невозможным (или, по крайней мере, чрезвычайно неудобным) писать программы, которые манипулируют блоками, на языке Pascal или любом другом жестко типизированном языке. Таким образом, в этом разделе мы вынуждены проявить непоследовательность и предлагать читателям только программы на псевдоPascal, поскольку о реальных программах на языке Pascal в данном случае не может быть и речи. Однако никаких проблем не возникнет, если те же алгоритмы реализовать на ассемблерных языках или языках системного программирования, таких как С.

Фрагментация и уплотнение пустых блоков

Чтобы проанализировать одну из специальных проблем, связанных с управлением динамической памятью, допустим, что переменная У, показанная на рис. 12.5, удалена; в результате блок, представляющий *Y*, необходимо вернуть в список свободного пространства. Проще всего вставить новый блок в начало списка свободного пространства, как показано на рис. 12.6. На этом рисунке показан пример ***фрагментации***, выражающейся в том, что крупные области памяти представляются в списке свободного пространства все более мелкими "фрагментами", т.е. множеством небольших блоков, составляющих целое (блок данных или свободное пространство). В рас-

[1] Обратите внимание: на рис. 12.1 счетчик показывает не длину блока, а длину данных.

сматриваемом случае, как видно из рис. 12.6, последние 2 300 байт динамической памяти пусты, тем не менее, все это пространство делится на три блока, содержащие 1 000, 600 и 700 байт соответственно, причем эти блоки даже не представлены в последовательном порядке (по местоположению в памяти) в списке свободного пространства. Поэтому в данном случае, не выполнив предварительно в той или иной форме "сборку мусора", удовлетворить запрос, например на блок 2 000 байт, не представляется возможным.

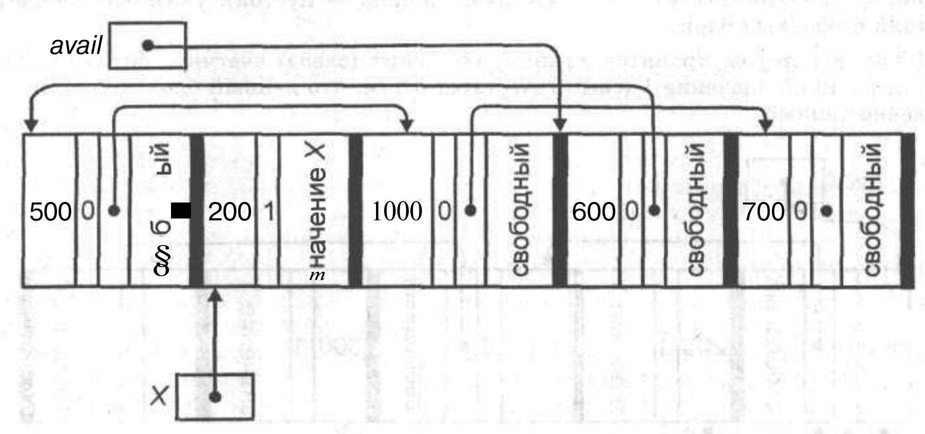


Рис. 12.6. Конфигурация памяти после освобождения блока переменной Y

Очевидно, что при возврате блока в список свободного пространства было бы желательно взглянуть на блоки, находящиеся непосредственно слева и справа от блока, который мы собираемся сделать доступным. Найти блок справа довольно просто. Если возвращаемый блок начинается с позиции p и имеет счетчик c, тогда блок справа начинается с позиции $p + c$. Если нам известно p, тогда остается только прочитать байты, начиная с позиции p, чтобы получить значение счетчика c. Начиная с байта $p + c$, пропускаем поле счетчика, чтобы найти бит, который говорит о том, является ли соответствующий блок пустым. Если этот блок пустой, тогда блоки, начинающиеся с р и p + с, можно объединить.

Пример 12.4. Допустим, что область динамической памяти, представленная на рис. 12.5, начинается с позиции 0. Тогда блок, возвращаемый переменной Y, начинается в байте **1700**, поэтому p = 1 700, а с = 600. Блок, начинающийся с позиции $p + c$ = 2 300, также пустой, поэтому можно объединить их в один блок, начинающийся с позиции 1 700, значением **счетчика** которого будет 1 300 (сумма значений счетчиков двух блоков). □

Однако откорректировать список свободного пространства после объединения блоков будет не так-то просто. Такой объединенный блок можно создать, просто добавив значение счетчика второго блока к значению c. Однако этот второй блок будет по-прежнему включен в список свободного пространства и его нужно оттуда убрать. Для этого потребуется найти указатель на этот блок у его предшественника в списке свободного пространства. Для этого предусмотрено несколько стратегий, ни одна из которых не имеет предпочтения перед другими.

1. Можно просматривать список свободного пространства до тех пор, пока не будет найден указатель, содержащий значение $p + c$. Этот указатель должен находиться в блоке-предшественнике того блока, который мы объединили с его соседом. Далее следует заменить найденный указатель на указатель, который находился в удаляемом блоке. Это приводит к удалению блока, начинающегося с позиции $p + c$, из списка свободного пространства. Разумеется, сам по себе этот блок будет по-прежнему доступен — просто теперь он является частью блока, начинаю-

щегося с позиции p. В среднем при использовании такого подхода приходится просматривать примерно половину списка свободного пространства, поэтому затраченное время будет пропорционально длине этого списка.

2. Можно использовать список свободного пространства с двойными связями. В этом случае возможно довольно быстро найти блок-предшественник и удалить из списка блок, начинающийся с позиции $p + c$. Для реализации этого подхода требуется постоянное время, не зависящее от длины списка свободного пространства, но он требует дополнительного пространства для хранения второго указателя в каждом пустом блоке, что приводит к увеличению минимального размера блока, который в этом случае должен содержать счетчик, бит заполнения и два указателя.

3. Можно хранить список блоков свободного пространства, отсортированный по позициям. В таком случае известно, что блок, начинающийся с позиции p, является предшественником блока, начинающегося с позиции $p + c$. Манипуляции с указателем, требующиеся для устранения второго блока, можно выполнять за фиксированное время. Однако вставка нового свободного блока требует просмотра в среднем половины списка свободного пространства, что делает этот метод ничуть не эффективнее первого метода.

Для возврата блока в список свободного пространства и объединения его со своим соседом справа (если этот сосед пустой) из перечисленных трех стратегий лишь методам (1) и (3) требуется время, пропорциональное длине списка свободного пространства. При выборе подходящей стратегии это время может оказаться решающим фактором — здесь все зависит от длины списка свободного пространства и от того, какая часть общего времени работы программы тратится на манипулирование динамической памятью. Второй метод — двойное связывание списка свободного пространства — обладает лишь одним недостатком: повышением минимального размера блоков. Кроме того, двойное связывание, подобно другим методам, мало помогает в поиске свободных соседей слева за время, меньшее того, которое требуется для просмотра списка свободного пространства.

Найти блок, расположенный непосредственно слева от интересующего нас блока, не так-то просто. Позиция p блока и его счетчик c, определяя позицию блока справа, не позволяют определить адрес блока, расположенного слева. Нам нужно найти пустой блок, который начинается в некоторой позиции p_1 и имеет счетчик c_1, причем $p_1 + c_1 = p$. Известны три варианта действий.

1. Можно просматривать список свободного пространства пока не будет найдет блок, который находится в позиции p_1 и имеет счетчик c_1, причем $p_1 + c_1 = p$. Выполнение этой операции занимает время, пропорциональное длине списка свободного пространства.

2. Можно поддерживать в каждом блоке (используемом или неиспользуемом) специальный указатель на позицию блока слева. Этот подход позволяет находить блок слева в течение фиксированного времени; затем можно проверить, является ли этот блок пустым, и, если это действительно так, объединить его с рассматриваемым нами блоком. Далее следует перейти в блок в позиции $p + c$ и сделать его указатель указывающим на начало нового блока, что позволит поддерживать функционирование таких "указателей влево".[1]

3. Можно поддерживать список блоков свободного пространства, отсортированный по позициям. Тогда при вставке в список вновь образовавшегося пустого блока легко найти пустой блок слева. Остается лишь проверить (зная позицию и счетчик "левого" пустого блока), нет ли между этими блоками каких-либо непустых блоков.

---

[1] Предлагаем читателю в качестве упражнения подумать над тем, как можно было бы обеспечить функционирование указателей, если блок расщепляется на два, при этом предполагается, что одна половина используется для хранения нового элемента данных, а вторая остается пустой.

Как и в. случае объединения вновь образовавшегося пустого блока с его соседом справа, первый и третий подходы к поиску и объединению с блоком слева требуют времени, пропорционального длине списка свободного пространства. Для реализации второго метода, как и в предыдущем случае, требуется фиксированное время, однако этому методу присущ недостаток, не имеющий отношения к проблемам, касающимся двойного связывания списка свободного пространства (о которых мы говорили в отношении поиска соседних блоков справа). В то время как двойное связывание пустых блоков повышает минимальный размер блока, нельзя сказать, что этот подход приводит к нерациональному расходованию памяти, поскольку в этом случае выполняется связывание лишь блоков, не используемых для хранения данных. Однако указание на соседей слева требует применения указателя как в используемых, так и в неиспользуемых блоках, и в этом случае действительно можно говорить о нерациональном расходовании памяти. Если средний размер блока составляет сотни байт, дополнительным местом для хранения указателя можно и пренебречь. В то же время, если длина стандартного блока равняется лишь 10 байт, дополнительное место для хранения указателя становится существенным фактором.

Обобщив наши рассуждения о том, каким способом следует объединять вновь образовавшиеся пустые блоков с их пустыми соседями, можно указать на три подхода к борьбе с фрагментацией.

1. Можно воспользоваться одним из нескольких подходов (например, хранение списка блоков свободного пространства в отсортированном виде), которые приводят к затратам времени, пропорциональным длине списка свободного пространства, каждый раз, когда блок становится неиспользуемым, но позволяют находить и объединять пустых соседей.
2. Можно воспользоваться списком блоков свободного пространства с двойной связью каждый раз, когда блок становится неиспользуемым; кроме того, можно применять указатели на соседа слева во всех блоках (используемых и неиспользуемых) для объединения за фиксированное время пустых соседей.
3. Для объединения пустых соседей ничего не делать в явном виде. Когда нельзя найти блок, достаточно большой, чтобы в нем можно было запомнить новый элемент данных, нужно просмотреть блоки слева направо, подсоединяя пустых соседей, а затем создавая новый список свободного пространства. Эскиз программы, реализующей такой алгоритм, показан в листинге 12.5.

Листинг 12.5. Объединение смежных пустых блоков

```
(1) procedure merge;
    var
(2)     p, q: указатели на блоки;
          { p указывает на левый конец объединенного пустого блока,
              q указывает на блок, расположенный справа от блока
              с указателем p, и который включается в объединенный блок,
              если этот блок справа пустой }
    begin
(3)     p:= крайний слева блок динамической памяти;
(4)     сделать список блоков свободного пространства пустым;
(5)     while p < правый конец динамической памяти do
(6)         if P указывает на полный блок со счетчиком c then
(7)             p:= p + c { пропуск заполненных блоков }
(8)         else begin { p указывает на начало последовательности
                            пустых блоков; их объединение }
(9)             q:= p + c { вычисление указателя q следующего блока }
(10)        while q указывает на пустой блок со счетчиком d и
                    q < правого конца динамической памяти do begin
```

```
(11)                добавить d к счетчику блока, указываемому p;
(12)                    q:= д + d
                    end;
(13)            вставить блок, на который указывает p, в список блоков
                    свободного пространства;
(14)            p:= g
            end
    end; { merge }
```

Пример 12.5. Применим программу из листинга 12.5 к схеме динамической памяти, изображенной на рис. 12.6. Допустим, что значение крайнего слева байта динамической памяти равно 0, поэтому вначале $p = 0$. Поскольку для первого блока $c = 500$, при вычислении q имеем $p + c = 500$. Поскольку блок, начинающийся с 500, заполнен, цикл, включающий строки (10) - (12), не выполняется, и блок, состоящий из байтов 0 - 499, подсоединяется к списку свободного пространства, в результате чего ячейка *avail* указывает на байт О, а в соответствующее место этого блока помещается указатель nil (непосредственно после счетчика и бита заполнения). Затем в строке (14) p присваивается значение 500, а в строке (7) р увеличивается до 700. Указателю q в строке (9) присваивается значение 1 700, а затем в строке (12) — 2 300 и 3 000, в то время как к значению счетчика 1 000 в блоке, начинающемся с 700-го байта, прибавляются 600 и 700. Когда q выйдет за пределы крайнего справа байта (2 999), блок, начинающийся с 700-го байта (значение счетчика в нем равно 2300), вставляется в список свободного пространства. Затем в строке (14) p присваивается значение 3 000 и внешний цикл завершается на строке (5). □

Когда общее количество блоков и количество свободных блоков отличаются не очень значительно, а вероятность случая, когда не удается обнаружить достаточно крупный пустой блок, относительно невелика, можно считать, что метод (3) — объединение смежных пустых блоков лишь в тех случаях, когда испытывается нехватка пространства, — предпочтительнее метода (1). Метод (2) считается возможным конкурентом указанных двух методов, но если принять во внимание его потребность в дополнительной памяти, а также тот факт, что каждый раз, когда блок вставляется в список блоков свободного пространства или удаляется оттуда, требуется некоторое дополнительное время для реорганизации списка, то можно прийти к выводу, что метод (2) предпочтительнее метода (3) в сравнительно редких случаях и в принципе можно вообще забыть о нем.

Выбор свободных блоков

Мы подробно обсудили, что следует делать, если какой-то блок данных освободился и его можно вернуть в список свободного пространства. Должен быть, однако, предусмотрен и обратный процесс — извлечение свободных блоков для запоминания в них новых данных. Очевидно, в этом случае надо выбрать тот или иной свободный блок и использовать его (целиком или частично) для запоминания в нем нового элемента данных. Но в этой ситуации необходимо решить два вопроса. Во-первых, какой именно пустой блок следует выбрать? Во-вторых, если возможно использовать только часть выбранного блока, то какую именно его часть следует использовать?

Второй вопрос решается достаточно просто. Если выбирается блок со счетчиком c и из этого блока требуется $d < c$ байт памяти, то используются последние d байт. В таком случае понадобится только заменить счетчик c на $c - d$, а оставшийся пустой блок может, как и ранее, оставаться в списке свободного пространства.[1]

Пример 12.6. Допустим, требуется 400 байт для переменной W в ситуации, представленной на рис. 12.6. Можно было бы принять решение изъять 400 байт из 600 в первом блоке списка свободного пространства. Ситуация в таком случае соответствовала бы рис. 12.7. D

---

[1] Если значение $c - d$ столь мало, что счетчик и указатель заполнения не умещаются в оставшееся свободное пространство, то, очевидно, надо использовать весь этот блок и убрать его из списка блоков свободного пространства.

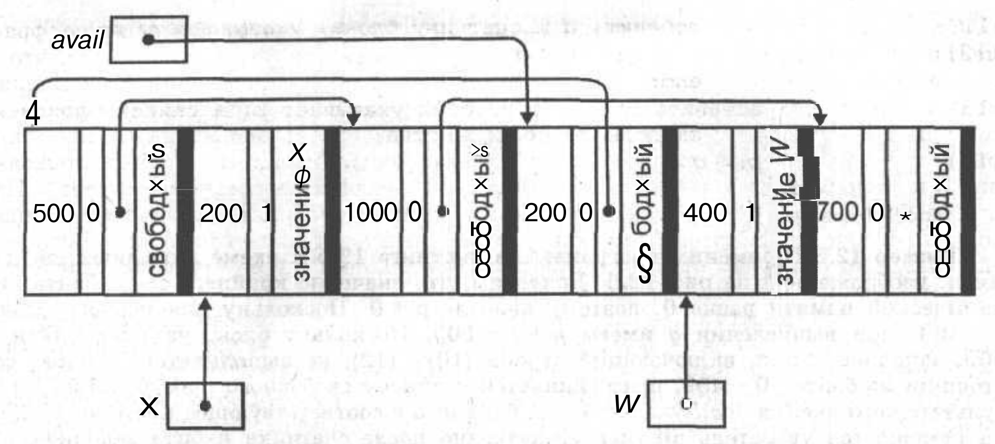


Рис. 12.7. Конфигурация памяти

Проблема выбора блока для размещения в нем новых данных не так уж проста, как может показаться, поскольку такие стратегии характеризуются противоречивыми целями. С одной стороны, мы хотим быстро выбрать пустой блок для размещения в нем новых данных, а с другой — хотим выбрать этот пустой блок таким образом, чтобы минимизировать фрагментацию. Две стратегии, которые представляют реализацию противоположных требований, называются "первый подходящий" (first-fit) и "самый подходящий" (best-fit). Описание этих стратегий приведено ниже.

1. *Первый подходящий.* В соответствии с этой стратегией, если требуется блок размера d, нужно просматривать список блоков свободного пространства с самого начала и до тех пор, пока не встретится блок размера $c \geq d$. Затем нужно использовать последние d байт (или слов) этого блока, как было описано выше.
2. *Самый подходящий.* В соответствии с этой стратегией, если требуется блок размера d, нужно проанализировать весь список блоков свободного пространства и найти блок размера не менее d, причем размер этого блока должен как можно меньше превышать величину d. Затем нужно использовать последние d байт этого блока.

По поводу этих стратегий можно высказать ряд соображений. Стратегия "самый подходящий" работает значительно медленнее, чем стратегия "первый подходящий", поскольку при использовании последней можно рассчитывать на достаточно быстрое (в среднем) нахождение подходящего блока, в то время как при использовании стратегии "самый подходящий" необходимо просмотреть весь список блоков свободного пространства. Стратегию "самый подходящий" можно несколько ускорить, если создать несколько списков блоков свободного пространства в соответствии с размерами этих блоков. Например, можно было бы иметь списки блоков размером от 1 до 16 байт,[1] от 17 до 32 байт, от 33 до 64 байт и т.д. Это "усовершенствование" не оказывает заметного влияния на быстродействии стратегии "первый подходящий", более того, оно может даже замедлить ее работу, если распределения размеров и размещения блоков неблагоприятны.[2] В свете последнего замечания мы можем определить некий спектр стратегий между "первым подходящим" и "самым подходящим", отыскивая "самый подходящий" среди первых k свободных блоков при некотором фиксированном значении k.

---

[1] Вообще говоря, существует понятие минимального размера блока (этот размер, очевидно, больше 1), поскольку блоки, если они входят в состав свободного пространства, должны содержать указатель на следующий блок, счетчик и бит заполнения.

[2] Например, подходящий блок находится в начале общего списка свободных блоков, но в конце нужного списка блоков, разбитых по размерам. — *Прим. ред.*

Стратегия "самый подходящий", по-видимому, способствует уменьшению фрагментации в сравнении со стратегией "первый подходящий" — в том смысле, что в стратегии "самый подходящий" просматривается тенденция к созданию чрезвычайно малых "фрагментов", т.е. остатков блоков. В то время как количество этих фрагментов примерно такое же, как и в случае стратегии "первый подходящий", они занимают, как правило, довольно небольшой объем. Однако стратегия "самый подходящий" не обнаруживает тенденции к созданию "фрагментов среднего размера". Напротив, свободные блоки оказываются либо очень малыми фрагментами, либо большими блоками. Поэтому возможны такие последовательности запросов, которые способна удовлетворить стратегия "первый подходящий" и не способна удовлетворить стратегия "самый подходящий", и наоборот.

Пример 12.7. Допустим, в соответствии с рис. 12.6, список свободного пространства состоит из блоков размерами 600, 500, 1 000 и 700 байт (в указанной последовательности). Если используется стратегия "первый подходящий" и сделан запрос на блок размером 400, то получим этот блок из блока размером 600, который оказался первым в списке и способен вместить блок заданного размера. Теперь, после выделения блока в 400 байт под данные, список свободного пространства состоит из блоков размерами 200, 500, 1 000 и 700 байт. В этой ситуации мы не в состоянии удовлетворить тотчас же три запроса на блоки размером 600 (хотя это было бы возможным после объединения смежных пустых блоков и/или перемещения использованных блоков в динамической памяти).

Если используется стратегия "самый подходящий" применительно к списку свободных блоков размера 600, 500, 1 000 и 700 байт и в систему поступил запрос на блок размером 400, то в соответствии с этой стратегией выбирается самый подходящий для такого запроса блок размером 500 байт, а список блоков свободного пространства принимает вид 600, 100, 1 000 и 700 байт. В этом случае можно было бы удовлетворить три запроса на блоки размером 600, не прибегая к какой-либо реорганизации памяти.

С другой стороны, бывают ситуации (взять хотя бы наш список 600, 500, 1 000 и 700 байт), когда стратегия "самый подходящий" не оправдывает себя, в то время как стратегия "первый подходящий" позволяет получить требуемый результат, не прибегая к реорганизации памяти. Допустим, в систему поступил запрос на блок размером 400 байт. Стратегия "самый подходящий", как и раньше, оставит список блоков размера 600, 100, 1 000 и 700 байт, тогда как стратегия "первый подходящий" оставит список блоков размера 200, 500, 1 000 и 700 байт. Допустим, что после этого в систему поступил запрос на блоки размером 1 000 и 700 байт; любая из стратегий выделила бы полностью два последних пустых блока, оставив в случае стратегии "самый подходящий" свободными блоки размером 600 и 100 байт, а в случае стратегии "первый подходящий" — блоки размером 200 и 500 байт. После этого стратегия "первый подходящий" может удовлетворить запросы на блоки размером 200 и 500 байт, в то время как стратегия "самый подходящий" — нет. □

12.5. Методы близнецов

Разработано целое семейство стратегий управления динамической памятью, которое позволяет частично решить проблему фрагментации и неудачного распределения размеров пустых блоков. Эти стратегии, называемые *методами близнецов* (buddy systems), на практике затрачивают очень мало времени на объединение смежных пустых блоков. Недостаток метода близнецов заключается в том, что блоки имеют весьма ограниченный набор размеров, поэтому, возможно, придется нерационально расходовать память, помещая элемент данных в блок большего размера, чем требуется.

Главная идея, лежащая в основе всех методов близнецов, заключается в том, что все блоки имеют лишь строго определенные размеры $s_1 < s_2 < s_3 < \dots < s_k$. Характерными вариантами последовательности чисел $s_1, s_2, \dots$ являются числа 1, 2, 4, 8, ... *(метод близнецов экспоненциального типа)* и 1, 2, 3, 5, 8, 13, ... *(метод близнецов с*

числами Фибоначчи, где $s_{i+1} = s_i + s_{i-1}$). Все пустые блоки размера s_i связаны в список; кроме того, существует массив заголовков списков свободных блоков, по одному для каждого допустимого размера s_i.[1] Если для нового элемента данных требуется блок размером d, то выбирается свободный блок такого размера s_i, чтобы $s_i > d$, однако $s_{i-1} < d$, т.е. наименьший допустимый размер, в который умещается новый элемент данных.

Трудности возникают в том случае, когда не оказывается пустых блоков требуемого размера s_i. В этом случае находится блок размером s_{i+1} и расщепляется на два блока, один размером s_i, а другой — $s_{i+1} - s_i$.[2] Метод близнецов налагает ограничение: величина $s_{i+1} - s_i$ должна равняться некоторому числу s_j из используемой последовательности, $j < i$. Теперь становится очевидным способ ограничения выбора значений для s_i. Если допустить, что $j — i - k$ для некоторого $k > 0$, тогда, поскольку $s_{i+1} - s_i = s_{i-k}$, следует, что

$$s_{i+1} = s_i + s_{i-k} \,. \tag{12.1}$$

Уравнение (12.1) применимо, когда $i > k$, и вместе со значениями $s_1, s_2, \ldots s_k$ полностью определяет значения $s_{k+1}, s_{k+2}, \ldots$. Если, например, $k = 0$, уравнение (12.1) принимает вид

$$s_{i+1} = 2s_i \,. \tag{12.2}$$

Если в (12.2) начальное значение $s_1 = 1$, то получаем экспоненциальную последовательность 1, 2, 4, 8. . . . Конечно, с какого бы значения s_1 мы ни начали, s_i в (12.2) возрастают экспоненциально. Еще один пример: если $k = 1$, $s_1 = 1$ и $s_2 = 2$, уравнение (12.1) принимает вид

$$s_{i+1} = s_i + s_{i-1}. \tag{12.3}$$

Уравнение (12.3) определяет последовательность *чисел Фибоначчи:* 1, 2, 3, 5, 8, 13, ...

Какое бы значение k в уравнении (12.1) мы ни выбрали, получим *метод близнецов k-го порядка.* Для любого k последовательность допустимых размеров возрастает по экспоненциальному закону, т.е. отношение s_{i+1}/s_i близко некоторой константе, большей единицы. Например, для $k = 0$ отношение s_{i+1}/s_i равно точно 2. Для $k = 1$ коэффициент s_{i+1}/s_i примерно равен "золотому сечению" ($(\sqrt{5}+1)/2 = 1{,}618$); этот коэффициент уменьшается при увеличении k, но никогда не достигает значения 1.

Распределение блоков

При использовании метода близнецов fe-го порядка можно считать, что каждый блок размером s_{i+1} состоит из блока размером s_i и блока размером s_{i-k}. Допустим для определенности, что блок размером s_i находится слева (в позициях с меньшими значениями номеров) от блока размером s_{i-k}.[3] Если динамическую память рассматривать как единый блок размером s_n (при некотором большом значении n), тогда позиции, с которых могут начинаться блоки размером s_i, будут полностью определены.

Позиции в экспоненциальном (т.е. 0-го порядка) методе близнецов вычислить достаточно просто. Если предположить, что позиции в динамической памяти пронумерованы с О, тогда блок размером s_i начинается в любой позиции с числа, кратного

[1] Поскольку в пустых блоках должны быть указатели на следующий блок (а также и другая информация), на самом деле последовательность допустимых размеров начинается не с 1, а с большего числа в указанной последовательности, например с 8 байт.

[2] Разумеется, если в системе отсутствуют и пустые блоки размером s_{i+1}, то создается такой блок путем расщепления блока размером s_{i+2}, и т.д. Если в системе вообще отсутствуют блоки более крупного размера, тогда, по сути, имеем дело с нехваткой памяти и необходимо реорганизовать динамическую память (о том, как это делается, рассказано в следующем разделе).

[3] Заметим, что блоки размерами s_i и s_{i-k}, составляющие блок размером s_{i+1}, удобно представлять "близнецами" (или "двойниками") — именно отсюда и происходит термин "метод близнецов".

2^i, т.е. O, 2^i. . . . Более того, каждый блок размером 2^{i+1}, начинающийся, скажем, с $j2^{i+1}$, состоит из двух "близнецов" размером 2^i, которые начинают с позиций $(2j)2^i$, т.е. $j2^{i+1}$, и $(2j + 1)2^i$. Таким образом, не составляет особого труда найти "близнеца" блока размером 2^i. Если он начинается с какого-либо четного числа, кратного 2^i, например $(2j)2^i$, его "близнец" находится справа в позиции $(2j + 1)2^i$. Если же он начинается с какого-либо нечетного числа, умноженного на 2^i, например $(2j + 1)2^i$, его "близнец" находится слева на позиции $(2j)2^i$.

Пример 12.8. Дело несколько усложняется, если речь идет о методе близнецов для порядка, большего, чем 0. На рис. 12.8 показано применение метода близнецов с числами Фибоначчи для динамической памяти размером 55 с блоками размеров $s_1, s_2, \ldots, s_8$ = 2, 3, 5, 8, 13, 21, 34 и 55. Например, блок размером 3, начинающийся с позиции 26, является близнецом блока размером 5, начинающегося с позиции 21; вместе они составляют блок размером 8, который начинается с позиции 21 и является близнецом блока размером 5, начинающегося с позиции 29. Вместе они составляют блок размером 13, начинающийся с позиции 21, и т.д. □

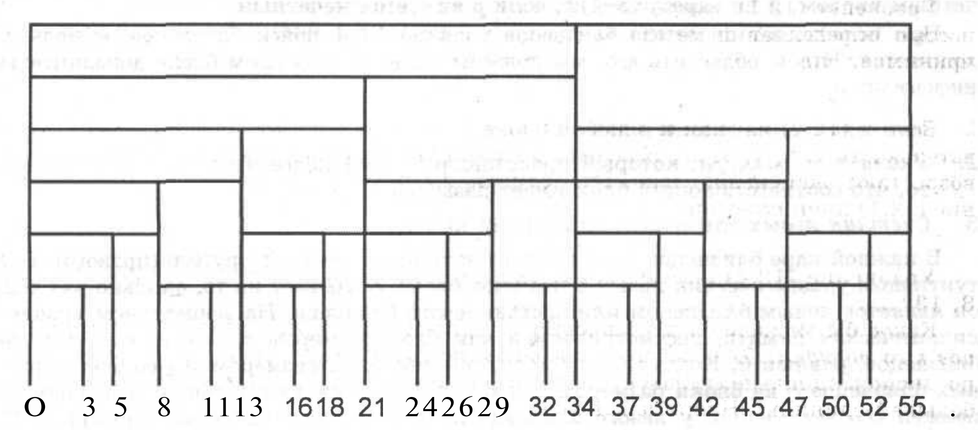


Рис. 12.8. Разделение динамической памяти в соответствии с методом близнецов с числами Фибоначчи

Выделение блоков

ЕСЛИ требуется блок размером n, мы выбираем любой из блоков, имеющихся в списке блоков свободного пространства размера s_i, где $s_i > n$, и либо $i = 1$, либо $s_{i-1} < n$; т.е. выбирается "самый подходящий" блок. При использовании метода близнецов ft-го порядка, если отсутствуют свободные блоки размером s_i, можем принять решение расщепить блок размером s_{i+1} или s_{i+k+1}, поскольку один из результирующих блоков будет в любом случае иметь размер s_i. Если нет блоков ни одного и из этих размеров, можно создать требуемый блок, применив рекурсивно описанную стратегию расщепления для размера s_{i+1}.

Однако здесь кроется небольшая ловушка. При использовании метода близнецов k-го порядка нельзя расщепить блоки размеров $s_1, s_2, \ldots s_k$, поскольку в результате получится блок, размер которого окажется меньше, чем s_1. Если у нас в распоряжении нет блока меньшего размера, придется использовать целый блок, не прибегая к его расщеплению. Эта проблема не возникает, если k = O, т.е. в случае экспоненциального метода близнецов. Решение этой проблемы можно упростить при использовании метода близнецов с числами Фибоначчи, если начать с $s_1 = 1$, но такой вариант может оказаться неприемлемым, поскольку блоки единичного размера (например, величиной в один байт или машинное слово) могут оказаться слишком малы для размещения в них указателя и бита заполнения.

Возврат блоков в свободное пространство

Когда блок становится доступным для повторного использования, проявляется одно из преимуществ метода близнецов. Иногда удается сократить фрагментацию, объединяя вновь образовавшийся свободный блок с его "близнецом", если этот "близнец" также **свободен.**[1] В сущности, если это действительно окажется так, результирующий блок можно попытаться объединить с его "близнецом" — если этот "близнец" также свободен, и т.д. Такое объединение с пустыми "близнецами" занимает строго фиксированное время и потому является привлекательной альтернативой периодическим слияниям смежных пустых блоков, о которых шла речь в **предыдущем** разделе (для выполнения такого слияния требуется время, пропорциональное количеству пустых блоков).

Применение экспоненциального метода близнецов чрезвычайно облегчает задачу обнаружения близнецов. Если мы **только** что вернули блок размером 2^i, начинающийся с позиции $p2^i$, его "близнец" находится по адресу $(p+1)2^i$, если p является четным числом, и по адресу $(p-1)2^i$, **если** p является нечетным.

При использовании метода близнецов порядка $k \geq 1$ поиск близнецов несколько усложняется. Чтобы облегчить его, мы должны хранить в каждом блоке дополнительную информацию.

1. Бит заполнения, как и в любом блоке.
2. *Указатель размера,* который представляет собой целое число i, указывающее на то, что соответствующий блок имеет размер s_i.
3. *Счетчик левых близнецов,* описанный ниже.

В каждой паре близнецов один (левый) находится слева от другого (правого). На интуитивном уровне счетчик левых близнецов блока указывает на то, сколько раз подряд он является левым близнецом или частью левого близнеца. На формальном уровне вся динамическая память, рассматриваемая как блок размером s_n, имеет счетчик левых **близнецов,** равный 0. Когда мы делим какой-либо блок размером s_{i+1} со счетчиком левых **близнецов** b на блоки размером s_i и s_{i-k}, которые являются левым и правым близнецами соответственно, у левого близнеца значение счетчика левых близнецов будет равно $b + 1$, в то время как у правого счетчик левых близнецов будет равен О и не зависеть, таким образом, от b. Например, на рис. 12.8 у блока размером 3, начинающегося с позиции О, счетчик левых близнецов равен 6, а у блока размером 3, начинающегося с позиции 13, счетчик левых близнецов равен 2.

Помимо указанной выше информации, пустые блоки (но не используемые блоки) содержат указатели вперед и назад для организации списка свободных блоков соответствующего размера. Эти двунаправленные указатели существенно облегчают объединение близнецов, так как в результате объединения требуется удалить из соответствующих списков объединившиеся блоки.

Эта информация используется следующим образом. Пусть используется метод близнецов порядка k. Любой блок, начинающийся с позиции p со счетчиком левых близнецов, равным О, является правым близнецом. Таким образом, если его указатель размера равен j, его левый близнец имеет размер s_{j+k} и начинается с позиции $p - s_{j+k}$. Если счетчик левых близнецов оказывается больше 0, значит, данный блок является левым близнецом блока размером s_{j-k}, который начинается с позиции $p + Sj$.

Если объединим левого близнеца размером Sj, имеющего счетчик левых близнецов, равный b, с правым близнецом размером s_{i-k}, результирующий блок будет иметь указатель размером i + 1, начинаться с той же позиции, что и блок размера *St,* и иметь счетчик левых близнецов, равный $b - 1$. Таким образом, когда объединяются два пустых близнеца, изменение служебной информации не вызывает затруднений.

---

[1] Как и в предыдущем разделе, мы должны предположить, что один бит каждого блока зарезервирован в качестве индикатора заполнения, указывающего, является ли соответствующий блок пустым или используется для **хранения** данных.

Предлагаем читателю самостоятельно рассмотреть ситуацию, когда расщепляется пустой блок размером s_{i+1} на используемый для хранения данных блок размером s_i и пустой блок размером s_{i-k}.

Если эта информация будет поддерживаться при реорганизации блоков и списки свободных блоков будут связаны в обоих направлениях, то потребуется лишь фиксированное время на каждое расщепление блока на близнецов или объединение близнецов в более крупный блок. Поскольку количество объединений не может превосходить количество расщеплений, общий объем работы будет пропорционален количеству расщеплений. Нетрудно убедиться в том, что большинство запросов на выделение блока вообще не требуют расщеплений, поскольку блок нужного размера уже имеется в наличии. Однако бывают нестандартные ситуации, когда каждая операция выделения требует нескольких расщеплений. Примером такой "нештатной" (которую можно даже назвать экстремальной) ситуации являются многократно повторяющиеся запросы на получение блока минимального размера с последующим его возвращением. В этой ситуации, если в системе динамической памяти имеются блоки n различных размеров, то при использовании метода близнецов k-го порядка потребуется по крайней мере n/k расщеплений, а затем еще n/k объединений при возврате блока.

12.6. Уплотнение памяти

Бывают случаи, когда даже после объединения всех смежных пустых блоков система не в состоянии выполнить запрос на выделение нового блока. Бывает, конечно, что во всей динамической памяти просто не хватает свободного пространства для выделения блока требуемого размера. Однако более типичной является ситуация, подобная той, которая показана на рис. 12.5, когда, несмотря на наличие 2 200 байт свободной памяти, мы не в состоянии выполнить запрос на выделение блока размером более 1 000 байт. Проблема заключается в том, что свободное пространство делится между несколькими несмежными блоками. Существуют два общих подхода к решению этой проблемы.

1. Пространство, выделяемое для хранения данных, состоит из нескольких пустых блоков. В таком случае можно потребовать, чтобы все блоки имели одинаковый размер и чтобы в них было предусмотрено место для указателя и данных. Указатель в используемом блоке указывает на следующий блок, используемый для хранения данных (в последнем блоке указатель равен нулю). Если бы, например, мы хранили данные, размер которых чаще всего невелик, то можно было бы выбрать блоки по 16 байт (4 — для указателя и 12 — для данных). Если же элементы данных обычно имеют большие размеры, следовало бы выбрать блоки размером несколько сотен байт (попрежнему 4 байта для указателя, а остальное для данных).
2. Если объединение смежных пустых блоков не позволяет получить достаточно большой блок, данные в динамической памяти нужно переместить таким образом, чтобы все заполненные блоки сместились влево (т.е. в сторону позиций с меньшими номерами); в этом случае справа образуется один крупный блок свободной памяти.

Метод (1) — использование цепочек блоков для хранения данных — как правило, приводит к перерасходу памяти. Если выбирается блок малого размера, это означает, что большая часть пространства будет использоваться для служебных целей (хранения указателей, с помощью которых формируется цепочка блоков). Если же используются крупные блоки, доля пространства, используемая для служебных целей, будет невелика, однако многие блоки все равно будут использоваться неэффективно при хранении в них небольших элементов данных. Единственной ситуацией, когда применение такого подхода представляется оправданным, является использование его для хранения очень больших элементов данных (если такие элементы данных являются типичными для вычислительной системы). Например, во многих современных файловых системах динамическая память (которая, как правило, размещается на дисках) делится на блоки одинакового размера, от 512 до 4096 байт, в

зависимости от конкретной системы. Поскольку размеры большинства файлов намного превышают эти величины, пространство используется достаточно эффективно, и указатели на блоки, составляющие файл, занимают относительно немного места. Выделение свободного пространства при использовании такого подхода выполняется по относительно простой схеме (учитывая то, о чем говорилось в предыдущих разделах), поэтому здесь мы не будем углубляться в рассмотрение этого вопроса.

Задача уплотнения памяти

Типичная задача, с которой постоянно приходится сталкиваться на практике, заключается в следующем: нужно совокупность используемых блоков (например, ту, которая показана на рис. 12.9, а), каждый из которых может иметь свой размер и на которых может указывать несколько указателей, смещать влево до тех пор, пока все свободное пространство не окажется с правой стороны динамической памяти, как показано на рис. 12.9, б. Указатели, конечно же, должны по-прежнему указывать на те же данные.



б. После уплотнения

Рис. 12.9. Процесс уплотнения памяти

У этой задачи есть несколько простых решений, если в каждом блоке предусмотреть небольшое дополнительное пространство. Но мы рассмотрим другой, более сложный метод. Этот метод весьма эффективен и не требует дополнительного места в используемых блоках (за исключением того пространства, которое требуется для любых уже обсуждавшихся нами схем управления памятью: бита заполнения и счетчика размера соответствующего блока).

Простая схема уплотнения заключается в том, что сначала нужно просмотреть все блоки слева направо (как заполненные, так и пустые) и вычислить так называемый "адрес передачи" (forwarding address) для каждого заполненного блока. Адрес передачи блока — это его текущая позиция минус сумма всего пустого пространства слева от него, т.е. позиция, в которую в конечном счете необходимо переместить этот блок. Адрес передачи блока вычислить несложно. Когда мы просматриваем все блоки слева направо, нужно суммировать объем встречающихся свободных блоков и вычитать этот объем из позиции каждого встречающегося блока. Эскиз соответствующего алгоритма представлен в листинге 12.6.

Листинг 12.6. Вычисление адреса передачи

```
(1)  var
         p: integer; { позиция текущего блока }
         gap: integer; { накапливающийся общий объем пустых блоков }
     begin
(2)      p:= левый край динамической памяти;
(3)      gap:= 0;
(4)      while p < правый край динамической памяти do begin
             { сделать p указателем на блок B }
(5)          if B — пустой блок then
(6)              gap:= gap + счетчик в блоке B
             else { B заполнен }
(7)              адрес передачи блока B := p - gap;
(8)          p:= p + счетчик в блоке B
         end
     end;
```

Вычислив адреса передачи, анализируем все указатели динамической **памяти.**[1] Следуя каждому указателю на некоторый блок *B,* этот указатель заменяется на адрес передачи, найденный для блока *B.* Наконец, все заполненные блоки перемещаются по их адресам передачи. Этот алгоритм подобен представленному в листинге 12.6, необходимо только заменить строку (7) на код

```
for i:= p to p - 1 + счетчик в блоке B do
    heap[i-gap]:= heap[i];
```

чтобы сместить блок *B* влево на величину **gap.**[2] Обратите внимание, что перемещение заполненных блоков занимает время, пропорциональное объему используемой динамической памяти, и, по-видимому, будет преобладать над другими временными затратами, связанными с уплотнением памяти.

Алгоритм Морриса

Ф. Л. Моррис (F. L. Morris) разработал метод уплотнения динамической памяти, не предусматривающий резервирования пространства в блоках для хранения адресов передачи. Ему, однако, требуется дополнительный бит метки конца цепочки указателей. Суть этого метода заключается в создании цепочки указателей, исходящей из определенной позиции в каждом заполненном блоке и связывающей все указатели с этим блоком. Например, на рис. 12.9, а мы видим три указателя, *A, D* и *E,* указывающих на крайний слева заполненный блок. На рис. 12.10 показана требуемая цепочка указателей. Часть пространства блока, равного размеру указателя, изъята из блока и помещена в конец цепочки, где находится указатель *A.*

Для создания таких цепочек указателей используется следующий метод. Сначала просматриваются все указатели в любом удобном порядке. Допустим, просматриваем указатель *p* на блок *B.* Если бит метки конца в блоке *B* равен 0, значит, *p* является первым найденным нами указателем на блок *B.* Мы помещаем в *p* содержимое тех позиций *B,* которые используются для цепочки указателей, и делаем так, чтобы эти позиции *B* указывали на *p.* Затем устанавливаем бит метки конца в блоке *B* в 1 (это означает, что теперь у него есть указатель), а бит

[1] Во всех действиях, которые следуют далее, мы полагаем, что совокупность этих указателей известна. Например, типичная реализация на языке **Snobol** запоминает пары, состоящие из имени переменной и указателя на значение этого имени в хеш-таблице, причем функция хеширования вычисляется на основе этого имени. Просмотр хеш-таблицы позволяет перебрать все указатели.

[2] В этом коде массив *heap* (куча) содержит адреса блоков динамической памяти. — *Прим. ред.*

метки конца в p устанавливаем в 0 (это означает конец цепочки указателей и присутствие смещенных данных).

Допустим теперь, что при просмотре указателя p на блок B бит метки конца в блоке B равен 1. Это значит, что блок B уже содержит заголовок цепочки указателей. Мы копируем в p указатель, содержащийся в B, и делаем так, чтобы B указывал на p, а бит метки конца в p устанавливаем в 1. Тем самым, по сути, вставляем p в заголовок цепочки указателей.



Рис. 12.10. Создание цепочки указателей

После того как мы свяжем все указатели на каждый блок в цепочку, исходящую из этого блока, можно переместить заполненные блоки как можно дальше влево (примерно так, как это сделано в рассмотренном выше алгоритме). Наконец, просматриваем каждый блок на его новой позиции и пробегаем всю его цепочку указателей. Теперь надо сделать так, чтобы каждый встреченный указатель указывал на блок в его новой позиции. Когда обнаруживается конец цепочки, переносим данные из блока B, содержащиеся в последнем указателе, в крайнее правое положение в блоке B и устанавливаем в 0 бит метки конца в этом блоке.

Упражнения

12.1. Рассмотрим следующий вариант динамической памяти из 1 000 байт, где "чистые" блоки используются в данный момент для хранения данных, а блоки, помеченные буквами, связаны в список свободной памяти в алфавитном порядке. Числа над блоками указывают первый байт в каждом блоке.

0	100	200	400	500	575	700	850	900	999
a		*b*		*c*	*d*		*e*	.	

Допустим, последовательно сделаны следующие запросы:

1) выделить блок в 120 байт;

2) выделить блок в 70 байт;

3) вернуть в начало списка свободного пространства блок, находящийся в позиции 700 - 849;

4) выделить блок в 130 байт.

Приведите списки свободного пространства (в соответствующей последовательности) после выполнения каждого запроса в предположении, что свободные блоки выбираются на основе стратегии

а) первый подходящий;

б) самый подходящий.

12.2. Рассмотрим следующий вариант динамической памяти, в котором "чистые" участки обозначают используемую память, а участки, помеченные буквами, — неиспользуемую, т.е. пустую.

O	100	200	300	500
	a		b	

Укажите последовательности запросов, которые можно удовлетворить, если использовать стратегии

а) первого подходящего, но не самого подходящего;

б) самого подходящего, но не первого подходящего.

*12.3. Допустим, что используется экспоненциальный метод близнецов с размерами блоков 1, 2, 4, 8 и 16 в динамической памяти размером 16. Если поступает запрос на блок размером n при $1 < n < 16$, то необходимо выделить блок размером 2^i, где $2^{i-1} < n < 2^i$. Неиспользуемую часть блока (если таковая имеется) нельзя использовать для удовлетворения любого другого запроса. Если нужен блок размером 2^i при $i < 4$ и такого свободного блока в динамической памяти нет, то сначала находится блок размером 2^{i+1} и расщепляется на две равные части. Если блока размером 2^{i+1} также нет, тогда сначала находится и расщепляется свободный блок размером 2^{i+2}, и т.д. Если оказывается, что необходим свободный блок размером 32, то следует признать, что удовлетворить поступивший запрос не удалось. В данном упражнении предполагается, что нельзя объединять смежные свободные блоки в динамической памяти.

Существуют последовательности запросов $a_1, a_2, \ldots, a_n$, сумма которых меньше 16, причем последний запрос в таких последовательностях удовлетворить невозможно. Рассмотрим, например, последовательность 5, 5, 5. Первый запрос приводит к расщеплению начального блока размером 16 на две части размером 8, причем одна из них используется для удовлетворения поступившего запроса. Оставшийся свободный блок размером 8 удовлетворяет второй запрос, но для удовлетворения третьего запроса свободного пространства уже нет.

Найдите последовательность $a_1, a_2, \ldots, a_n$ целых чисел (необязательно одинаковых) из интервала от 1 до 16, сумма которых по возможности минимальна, но если эту последовательность рассматривать как последовательность запросов на блоки размеров $a_1, a_2, \ldots, a_n$, то последний запрос удовлетворить невозможно. Поясните, почему эту последовательность запросов невозможно удовлетворить, а любую последовательность, сумма которой окажется меньше, можно.

12.4. Рассмотрим задачу уплотнения памяти для блоков одинакового размера. Допустим, каждый блок состоит из поля данных и поля указателя. Допустим также, что уже помечены все используемые в настоящее время блоки. Пусть эти блоки находятся в области памяти, расположенной между адресами a и b. Необходимо поменять местоположение всех блоков, занятых данными, так, чтобы они занимали непрерывный участок памяти, начиная с адреса a. Перемещая блок, следует помнить, что необходимо скорректировать поле указателя любого блока, указывающее на перемещаемый блок. Разработайте алгоритм уплотнения таких блоков.

12.5. Рассмотрим массив размера n. Разработайте алгоритм циклического сдвига против часовой стрелки всех элементов в таком массиве на k позиций, используя только фиксированную дополнительную память, объем которой не зависит от k и n. *Совет.* Проанализируйте, что произойдет, если переставить в обратном порядке первые k элементов, последние $n - k$ элементов и, наконец, весь массив.

12.6. Разработайте алгоритм замены вложенной строки символов y, принадлежащей строке xyz, на другую вложенную строку y', используя для этого как можно меньшее количество дополнительной памяти. Какова временная сложность (время выполнения) этого алгоритма и сколько для него необходимо памяти?

12.7. Напишите программу получения копии заданного списка. Какова временная сложность этого алгоритма и сколько для него необходимо памяти?

12.8. Напишите программу, которая проверяла бы идентичность двух списков. Какова временная сложность этого алгоритма и сколько для него необходимо памяти?

12.9. Реализуйте алгоритм Морриса для уплотнения динамической памяти (см. раздел 12.6).

*12.10. Разработайте схему выделения памяти применительно к ситуации, когда память выделяется и освобождается блоками длиной 1 и 2. Перечислите преимущества и недостатки такого алгоритма.

Библиографические примечания

Эффективное управление памятью — центральная проблема, которую решают разработчики многих языков программирования, в том числе Snobol [30], Lisp [74], APL [56] и SETL [98]. В [80] и [86] обсуждают методы управления памятью в контексте компиляции языков программирования.

Выделение памяти по методу близнецов было впервые описано в [62]. Метод близнецов с числами Фибоначчи рассмотрен в работе [48].

Изящный алгоритм маркировки, предназначенный для использования в процедуре чистки памяти, был разработан Питером Дойчем (Peter Deutsch) [25], а также Шорром и Уэйтом (Schorr and Waite) [97]. Схема уплотнения динамической памяти, изложенная в разделе 12.6, заимствована из статьи [77].

В работах [90] и [91] анализирует объем памяти, требующийся для работы алгоритмов динамического выделения памяти. В [92] представлен алгоритм ограниченного рабочего пространства (bounded workspace algorithm) для копирования циклических структур. Решение упражнения 12.5 можно найти в [32].

Список литературы

1. Адельсон-Вельский Г. М., Ландис Е. М. (1962). Один алгоритм организации информации. *Докл. АН СССР,* 146, с. 263-266.
2. Aho, A. V., M. R. Garey, and J. D. Ullman (1972). The transitive reduction of a directed graph, *SIAM J. Computing* 1:2, pp. 131-137.
3. Aho, A. V., J. E. Hopcroft, and J. D. Ullman (1974). *The Design and Analysis of Computer Algorithms,* Addison-Wesley, Reading, Mass. (Русский перевод: Ахо А., Хоркрофт Дж., Ульман Дж. Построение и анализ вычислительных алгоритмов. — М., "Мир", 1979.)
4. Aho, A. V., and N. J. A. Sloane (1973). Some doubly exponential sequences, *Fibonacci Quarterly,* 11:4, pp. 429-437.
5. Aho, A. V., and J. D. Ullman (1977). *Principles of Compiler Design,* Addison-Wesley, Reading, Mass.
6. Bayer, R., and E. M. McCreight (1972). Organization and maintenance of large ordered indices, *Acta Informatica* 1:3, pp. 173-189.
7. Bellman, R. E. (1957). *Dynamic Programming,* Princeton University Press, Princeton, N. J. (Русский перевод: Беллман Р. Динамическое программирование. — М., ИЛ, 1960.)
8. Bentley, J. L. (1982). *Writing Efficient Programs,* Prentice-Hall, Englewood Cliffs, N. J.
9. Bentley, J. L., D. Haken, and J. B. Saxe (1978). A general method for solving divide-and-conquer recurrences, CMU-CS-78-154, Dept. of CS, Carnegie-Mellon Univ., Pittsburg, Pa.
10. Berge, C. (1957). Two theorems in graph theory, *Proc. National Academy of Sciences* 43, pp. 842-844.
11. Berge, C. (1958). *The Theory of Graphs and its Applications,* Wiley, N. Y. (Русский перевод: Берж С. Теория графов и ее применение. — М., ИЛ, 1962.)
12. Birtwistle, G. M., O.-J. Dahl, B. Myhrhaug, and K. Nygaard (1973). *SIMULA Begin,* Auerbach Press, Philadelphia, Pa.
13. Blum, M., R. W. Floyd, V. R. Pratt, R. L. Rivest, and R. E. Tarjan (1972). Time bounds for selection, *J. Computer and System Sciences* 7:4, pp. 448-461.
14. Boruvka, 0. (1926). On a minimal problem, *Práce Moraské Pridovedecké Spolecnosti* 3:3, pp. 37-58.
15. Brooks, F. P. (1974). *The Mythical Man Month,* Addison-Wesley, Reading, Mass.
16. Carter, J. L., and M. N. Wegman (1977). Universal classes of hash functions, *Proc. Ninth Annual ACM Symp. on Theory of Computing,* pp. 106-112.
17. Cheriton, D., and R. E. Tarjan (1976). Finding minimum spanning trees, *SIAM J. Computing* 5:4, pp. 724-742.
18. Cocke, J., and F. E. Allen (1976). A program data flow analysis procedure, *Comm. ACM* 19:3, pp. 137-147.
19. Coffman, E. G. (ed.) (1976). *Computer and Job Shop Scheduling Theory,* John Wiley and Sons, New York.
20. Comer, D. (1979). The ubiquitous B-tree, *Computing Surveys* 11, pp. 121-137.
21. Cooley, J. M., and J. W. Tukey (1965). An algorithm for the machine calculation of complex Fourier series, *Math. Comp.* 19, pp. 297-301.
22. DBTG (1971). *CODASYL Data Base Task Group April 1971 Report,* ACM, New York.

23. Demers, A., and J. Donahue (1979). Revised report on RUSSELL, TR79-389, Dept. of Computer Science, Cornell Univ., Ithaca, N. Y.

24. Deo, N. (1975). *Graph Theory with Applications to Engineering and Computer Science,* Prentice-Hall, Englewood Cliffs, N. J.

25. Deutsch, L. P., and D. G. Bobrow (1966). An efficient incremental automatic garbage collector, *Comm. ACM* 9:9, pp. 522-526.

26. Dijkstra, E. W. (1959). A note on two problems in connexion with graphs, *Numerische Mathematik* 1, pp. 269-271.

27. Edmonds, J. (1965). Paths, trees, and flowers, *Canadian J. Math* 17, pp. 449-467.

28. Even, S. (1980). *Graph Algorithms,* Computer Science Press, Rockville, Md.

29. Even, S., and O. Kariv (1975). An $O(n^{2.5})$ algorithm for maximum matching in general graphs, *Proc. IEEE Sixteenth Annual Symp. on Foundations of Computer Science,* pp. 100-112.

30. Farber, D., R. E. Griswold, and I. Polonsky (1964). SNOBOL, a string manipulation language, *J. ACM* 11:1, pp. 21-30.

31. Fischer, M. J. (1972). Efficiency of equivalence algorithms, in *Complexity of Computer Computations* (R. E. Miller and J. W. Thatcher, eds.), pp. 153-168.

32. Fletcher, W., and R. Silver (1966). Algorithm 284: interchange of two blocks of data, *Comm. ACM* 9:5, p. 326.

33. Floyd, R. W. (1962). Algorithm 97: shortest path, *Comm. ACM 5:6,* p. 345.

34. Floyd, R. W. (1964). Algorithm 245: treesort 3, *Comm. ACM* 7:12, p. 701.

35. Floyd, R. W., and A. Smith (1973). A linear time two-tape merge, *Inf. Processing letters* 2:5, pp. 123-126.

36. Ford, L. R., and S. M. Johnson (1959). A tournament problem, *Amer. Math. Monthly* 66, pp. 387-389.

37. Frazer, W. D., and A. C. McKellar (1970). Samplesort: a sampling approach to minimal tree storage sorting, *J. ACM* 17:3, pp. 496-507.

38. Fredkin, E. (I960). Trie memory, *Comm. ACM* 3:9, pp. 490-499.

39. Friend, E. H. (1956). Sorting on electronic computer systems, *J. ACM* 3:2, pp. 134-168.

40. Fuchs, H., Z. M. Kedem, and S. P. Uselton (1977). Optimal surface reconstruction using planar contours, *Comm. ACM* 20:10, pp. 693-702.

41. Garey, M. R., and D. S. Johnson (1979). *Computers and Intractability: a Guide to the Theory of NP-Completeness,* Freeman, San Francisco. (Русский перевод: Гэри М., Джонсон Д.С. Вычислительные машины и трудноразрешимые задачи. — М., "Мир", 1982.)

42. Geschke, C. M., J. H. Morris, Jr., and E. H. Satterthwaite (1977). Early expreience with MESA, *Comm. ACM* 20:8, pp. 540-552.

43. Gotlieb, C. C., and L. R. Gotlieb (1978). *Data Types and Data Structures,* Prentice-Hall, Englewood Cliffs, N. J.

44. Greene, D. H., and D. E. Knuth (1983). *Mathematics for the Analysis of Algorithms,* Birkhauser, Boston, Mass. (Русский перевод: Грин Д., Кнут Д., Математические методы анализа алгоритмов. — М., "Мир", 1987.)

45. Gudes, E., and *S.* Tsur (1980). Experiments with B-tree reorganization, *ACM SIGMOD Symposium on Management of Data,* pp. 200-206.

46. Hall, M. (1948). Distinct representatives of subsets, *Bull. AMS* 54, pp. 922-926.

47. Harary, F. (1969). *Graph Theory,* Addison-Wesley, Reading, Mass. (Русский перевод: Харари Ф., Теория графов. — М., "Мир", 1973.)

48. Hirschberg, D. S. (1973). A class of dynamic memory allocation algorithms, *Comm. ACM* 16:10, pp. 615-618.

49. Hoare, C. A. R. (1962). Quicksort, *Computer J.* 5:1, pp. 10-15.

50. Hoare, C. A. R., O.-J. Dahl, and E. W. Dijkstra (1972). *Structured Programming*, Academic Press, N. Y. (Русский перевод: Дал У., Дейкстра Э., Хоор К. Структурное программирование. — М., "Мир", 1975.)

51. Hopcroft, J. E., and R. M. Karp (1973). An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs, *SIAMJ. Computing* 2:4, pp. 225-231.

52. Hopcroft, J. E., and R. E. Tarjan (1973). Efficient algorithms for graph manipulation, *Comm. ACM* 16:6, pp. 372-378.

53. Hopcroft, J. E., and J. D. Ullman (1973). Set merging algorithms, *SIAM J. Computing* 2:4, pp. 294-303.

54. Huffman, D. A. (1952). A method for the construction of minimum-redundancy codes, *Proc. IRE* 40, pp. 1098-1101.

55. Hunt, J. W., and T. G. Szymanski (1977). A fast algorithm for computing longest common subsequences, *Comm. ACM* 20:5, pp. 350-353.

56. Iverson, K. (1962). *A Programming Language*, John Wiley and Sons, New York.

57. Johnson, D. B. (1975). Priority queues with update and finding minimum spanning trees, *Inf. Processing Letters* 4:3, pp. 53-57.

58. Johnson, D. B. (1977). Efficient algorithms for shortest paths is sparse networks, *J. ACM* 24:1, pp. 1-13.

59. Карацуба А. А., Офман Ю. П. (1961). Умножение многозначных чисел на автоматах. *Докл. АН СССР*, 145, № 2, с. 293-294.

60. Kernighan, B. W., and P. J. Plauger (1974). *The Elements of Programming Style*, McGraw-Hill, N. Y.

61. Kernighan, B. W., and P. J. Plauger (1981). *Software Tools in Pascal*, Addison-Wesley, Reading, Mass.

62. Knowlton, K. C. (1965). A fast storage allocator, *Comm. ACM* 8:10, pp. 623-625.

63. Knuth, D. E. (1968). *The Art of Computer Programming Vol. I: Fundamental Algorithms*, Addison-Wesley, Reading, Mass. (Русский перевод: Кнут Д. Искусство программирования для ЭВМ. Том 1: Основные алгоритмы. — М., "Мир", 1976. Русский перевод переработанного издания: Кнут Д. Искусство программирования. Том 1: Основные алгоритмы. — М., Издательский дом "Вильяме", 2000.)

64. Knuth, D. E. (1971). Optimum binary search trees, *Acta Informatica* 1:1, pp. 14-25.

65. Knuth, D. E. (1973). *The Art of Computer Programming Vol. III: Sorting and Searching*, Addison-Wesley, Reading, Mass. (Русский перевод: Кнут Д. Искусство программирования для ЭВМ. Том 3: Поиск и сортировка. — М., "Мир", 1976. Русский перевод переработанного издания: Кнут Д. Искусство программирования. Том 3: Поиск и сортировка. — М., Издательский дом "Вильяме", 2000.)

66. Knuth, D. E. (1981). *TEX and Metafont, New Directions in Typesetting*, Digital Press, Bedford, Mass.

67. Kruskal, J. B. Jr. (1956). On the shortest spanning subtree of a graph and the traveling salesman problem, *Proc. AMS* 7:1, pp. 48–50.

68. Lin, S., and B. W. Kernighan (1973). A heuristic algorithm for the traveling salesman problem, *Operations Research* 21, pp. 498–516.

69. Liskov, B., A. Snyder, R. Atkinson, and C. Scaffert (1977). Abstraction mechanisms in CLU, *Comm. ACM* 20:8, pp. 564-576.

70. Liu, C. L. (1968). *Introduction to Combinatorial Mathematics*, McGraw-Hill, N. Y.

71. Lueker, G. S. (1980). Some techniques for solving recurrences, *Computing Surveys,* 12:4, pp. 419-436.

72. Lum, V., and H. Ling (1970). Multi-attribute retrieval with combined indices, *Comm. ACM* 13:11, pp. 660-665.

73. Maurer, W. D., and T. G. Lewis (1975). Hash table methods, *Computing Surveys* 7:1, pp. 5-20.

74. McCarthy, J. et al. (1965). *LISP 1.5 Programmers Manual.* MIT Press, Cambridge, Mass.

75. Micali, S., and V. V. Vazirani (1980). An $O(\sqrt{|V|} \cdot |E|)$ algorithm for finding maximum matching in general graphs, *Proc. IEEE Twenty-first Annual Symp. on Foundations of Computer Science,* pp. 17-27.

76. Moenck, R., and A. B. Borodin (1972). Fast modular transforms via division, *Proc. IEEE Thirteenth Annual Symp. on Switching and Automata Theory,* pp. 90-96.

77. Morris, F. L. (1978). A time- and space-efficient garbage compaction algorithm, *Comm. ACM* 21:8, pp. 662-665.

78. Morris, R. (1968). Scatter storage techniques, *Comm. ACM* 11:1, pp. 35-44.

79. Moyles, D. M., and G. L. Thompson (1969). Finding a minimum equivalent graph of a digraph, *J. ACM* 16:3, pp. 455-460.

80. Nicholls, J. E. (1975). *The Structure and Design of Programming Languages,* Addison-Wesley, Reading, Mass.

81. Nievergelt, J. (1974). Binary search trees and file organization, *Computer Surveys* 6:3, pp. 195-207.

82. Papadimitriou, C. H., and K. Steiglitz (1982). *Combinatorial Optimization: Algorithms and Complexity,* Prentice-Hall, Englewood Cliffs, N. J. (Русский перевод: Пападимитриу X., Стайглиц К. Комбинаторная оптимизация. Алгоритмы и сложность. — М., "Мир", 1985.)

83. Parker, D. S. Jr. (1980). Conditions for optimality of the Huffman algorithm, *SIAM J. Computing* 9:3, pp. 470-489.

84. Perl, Y., A. Itai, and H. Avni (1978). Interpolation search — a log log *n* search, *Comm.ACM* 21:7, pp. 550-553.

85. Peterson, W. W. (1957). Addressing for random access storage, *IBM J. Res. and Devel.* 1:2, pp. 130-146.

86. Pratt, T. W. (1975). *Programming Languages: Design and Implementation,* Prentice-Hall, Englewood Cliffs, N. J. (Русский перевод: Пратт Т. Языки программирования. Разработка и реализация. — М., "Мир", 1979.)

87. Pratt, V. R. (1979). *Shellsort and Sorting Networks,* Garland, New York.

88. Prim, R. C. (1957). Shortest connection networks and some generalizations, *Bell System Technical J.* 36, pp. 1389-1401.

89. Reingold, E. M. (1972). On the optimality of some set algorithms, *J. ACM* 19:4, pp. 649-659.

90. Robson, J. M. (1971). An estimate of the store size necessary for dynamic storage allocation, *J. ACM* 18:3, pp. 416-423.

91. Robson, J. M. (1974). Bounds for some functions concerning dynamic storage allocation, *J. ACM* 21:3, pp. 491-499.

92. Robson, J. M. (1977). A bounded storage algorithm for copying cyclic structures, *Comm. ACM* 20:6, pp. 431-433.

93. Sahni, S. (1974). Computationally related problems, *SIAM J. Computing* 3:3, pp. 262-279.

94. Sammet, J. E. (1968). *Programming Languages: History and Fundamentals,* Prentice-Hall, Englewood Cliffs, N. J.

95. Schkolnick, M. (1975). The optimal selection of secondary indices for files, *Information Systems* 1, pp. 141-146.

96. Schoenhage, A., and V. Strassen (1971). Schnelle multiplikation grosser zahlen, *Computing 7,* pp. 281-292.

97. Schorr, H., and W. M. Waite (1967). An efficient machine-independent procedure for garbage collection in various list structures, *Comm. ACM* 10:8, pp. 501-506.

98. Schwartz, J. T. (1973). *On Programming: An Interrum Report on the SETL Project,* Courant Inst., New York.

99. Sharir, M. (1981). A strong-connectivity algorithm and its application in data flow analysis, *Computers and Mathematics with Applications* 7:1, pp. 67-72.

100. Shaw, M., W. A. Wulf, and R. L. London (1977). Abstraction and verification in ALPHARD: defining and specifying iteration and generators, *Comm. ACM* 20:8, pp. 553-563.

101. Shell, D. L. (1959). A high-speed sorting procedure, *Comm. ACM* 2:7, pp. 30-32.

102. Shell, D. L. (1971). Optimizing the polyphase sort, *Comm. ACM* 14:11, pp. 713-719.

103. Singleton, R. C. (1969). Algorithm 347: an algorithm for sorting with minimal storage, *Comm. ACM* 12:3, pp. 185-187.

104. Strassen, V. (1969). Gaussian elimination is not optimal, *Numerische Mathematik* 13, pp. 354-356.

105. Stroustrup, B. (1982). Classes: an abstract data type facility for the C language, *SIGPLAN Notices* 17:1, pp. 354-356.

106. Suzuki, N. (1982). Analysis of pointer 'rotation', *Comm. ACM* 25:5, pp. 330-335.

107. Tarjan, R. E. (1972). Depth first search and linear graph algorithms, *SIAM J. Computing* 1:2, pp. 146-160.

108. Tarjan, R. E. (1975). On the efficiency of a good but not linear set merging algorithm, *J. ACM* 22:2, pp. 215-225.

109. Tarjan, R. E. (1981). Minimum spanning trees, неопубликованный меморандум, Bell Laboratories, Murray Hill, N. J.

110. Tarjan, R. E. (1983). *Data Structures and Graph Algorithms,* неопубликованная рукопись. Bell Laboratories, Murray Hill, N. J.

111. Ullman, J. D. (1974). Fast algorithms for the elimination of common subexpressions, *Acta Informatica* 2:3, pp. 191-213.

112. Ullman, J. D. (1982). *Principles of Database Systems,* Computer Science Press, Rockville, Md.

113. van Emde Boas, P., R. Kaas, and E. Zijlstra (1977). Design and implementation of an efficient priority queue structure, *Math Syst. Theory,* 10, pp. 99-127.

114. Warshall, S. (1962). A theorem on Boolean matrices, *J. ACM* 9:1, pp. 11-12.

115. Weinberg, G. M. (1971). *The Psychology of Computer Programming,* Van Nostrand, N. Y.

116. Weiner, P. (1973). Linear pattern matching algorithms, *Proc. IEEE Fourteenth Annual Symp. on Switching and Automata Theory,* pp. 1-11.

117. Wexelblat, R. L. (ed.) (1981). *History of Programming Languages,* Academic Press, N. Y.

118. Wiederhold, G. (1982). *Database Design,* McGraw-Hill, New York.

119. Williams, J. W. J. (1964). Algorithm 232: Heapsort, *Comm. ACM* 7:6, pp. 347-348.

120. Wirth, N. (1973). *Systematic Programming: An Introduction,* Prentice-Hall, Englewood Cliffs, N. J. (Русский перевод: Вирт Н. Системное программирование: Введение. — М., "Мир", 1977.)

121. Wirth, N. (1976). *Algorithms + Data Structures = Programs,* Prentice-Hall, Englewood Cliffs, N. J. (Русский перевод: Вирт Н. Алгоритмы + структуры данных = программы. — М., "Мир", 1985.)

122. Wulf, W. A., M. Shaw, P. Hilfinger, and L. Flon (1981). *Fundamental Structures of Computer Science,* Addison-Wesley, Reading, Mass.

123. Yao, A. C. (1975). An $O(|E| \log \log |V|)$ algorithm for finding minimum spanning trees, *Inf. Processing Letters* 4:1, pp. 21-23.

124. Yao, A. C., and F. F. Yao (1976). The complexity of searching a random ordered table, *Proc. IEEE Seventeenth Annual Symp. on Foundations of Computer Science,* pp. 173-177.

125. Yourdon, E., and L. L. Constantine (1975). *Structured Design,* Yourdon, New York.

Предметный указатель

Б

В

Г

Д

З

И

К

Л

М

Н

О

П

Р

С

Т

У

Ф

Х

Ц

Ч

Э

Я

Альфред В. Ахо, Джон Хопкрофт, Джеффри Д. Ульман

Структуры данных и алгоритмы

Литературный редактор *И. А. Попова*
Верстка *М. А. Удалое*
Художественный редактор *В. Г. Павлютин*
Технический редактор *Г. Н. Горобец*
Корректоры *З. В. Александрова, О. В. Мишутина, Л. В. Пустовойтова*

Издательский дом "Вильяме".
101509, Москва, ул. Лесная, д. 43, стр. 1.
Изд. лиц. ЛР N° 090230 от 23.06.99
Госкомитета РФ по печати.

Подписано в печать 21.05.2003. Формат 70X100/16.
Гарнитура SchoolBook. Печать офсетная.
Усл. печ. л. 30,96. Уч.-изд. л. 27,22.
Доп. тираж 3000 экз. Заказ № 48.

Отпечатано с диапозитивов в ФГУП "Печатный двор"
Министерства РФ по делам печати,
телерадиовещания и средств массовых коммуникаций.
197110, Санкт-Петербург, Чкаловский пр., 15.